(4)

BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 7047

Research and Development in Natural Language Understanding as Part of the Strategic Computing Program

Annual Technical Report
December 1987 - December 1988

Lance A. Ramshaw

Submitted to:
Defense Advanced Research Projects Agency

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Distribution of this document is unlimited. It may be released to the Clearinghouse, Dept. of Commerce, for sale to the general public.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) BBN Report No. 7047	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION BBN Systems and Technologies Corporation 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research
6c ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge, MA 02138	7b ADDRESS (City, State, and ZIP Code) Department of the Navy Arlington, VA 22217
8a NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Projects Agency 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-C-0016
8c ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22209	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Research and Development in Natural Language Understanding as Part of the Strategic Computing Program: Annual Technical Report Dec. 1987 - Dec. 1988

12 PERSONAL AUTHOR(S)
Lance A. Ramshaw

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Annual Report	FROM 12/87 TO 12/88	1989, July	186

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	09	
05	07	

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Natural language processing; plan recognition; metaplans; illformed input; robustness; artificial intelligence; discourse; (KT)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes in depth on area of the research conducted at BBN during 1988 as part of DARPA's Strategic Computing Program. This work explored the use of a model of the plans and goals of the user of a computer system to increase the robustness of the system's natural language understanding component.

Many examples of ill-formedness that people routinely and easily correct can be resolved by a natural language system only if it makes use of knowledge of the pragmatic context. This investigation centers around examples of alias errors, where the ill-formedness is due to a single word that is incorrect but still lexically understood, as with the substitution of on for in in the phrase stay on good shape. Localizing and resolving such errors frequently depends on pragmatic knowledge. This thesis presents a model for pragmatic context within expert advising dialogues, where an agent who is building a plan to solve a problem consults with a domain expert, and develops methods for applying that model to resolving ill-formed input.

cont'd...

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. Abstract (cont'd)

Metaplans are used to model the structure of the agent's problem-solving behavior, both the gradual refinement of the domain plans being considered and the connection between them and the queries motivated by them. The partially-specified domain plans that the agent is considering are represented by nodes in a plan classification hierarchy, and these classes of domain plans in turn serve as arguments to the problem-solving metaplans. The expansion and search of the metaplan tree that models the problem-solving context is governed by heuristics based both on its metaplan structure and on a model of the agent's world knowledge. This model can be used to track the problem-solving moves implicit in a sequence of well-formed queries and also to predict likely moves and queries as determined by the context which can then be linked to the partial interpretation of an ill-formed query suggesting corrections for the ill-formedness.

This approach has been implemented in a system called Pragma, which suggests corrections based on pragmatic context for alias errors in naval domain queries, using techniques that could also be extended to other classes of ill-formedness and to generating cooperative responses. Pragma demonstrated that a model capturing the pragmatic structure of a particular discourse setting can be used to increase the robustness of a natural language interface.

cont'd previous page

RESEARCH AND DEVELOPMENT IN NATURAL LANGUAGE UNDERSTANDING AS PART OF THE STRATEGIC COMPUTING PROGRAM

Annual Technical Report
21 December 1987 - 20 December 1988

Lance A. Ramshaw

Principal Investigator:
Dr. Ralph M. Weischedel

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

ARPA Order No. 5257

Contract No. N00014-85-C-0016

Scientific Officer:
Dr. Alan R. Meyrowitz

PREFACE

This annual report focuses on a single issue addressed under this contract during 1988. It therefore represents an in-depth study of one of the research goals of the research and development, rather than providing a broad, more shallow sampling of the work performed.

This report was also submitted as a Ph.D. dissertation to the University of Delaware Department of Computer and Information Sciences. It is also available from the University as Technical Report 89-18.

Pragmatic Knowledge for Resolving Ill-Formedness

TABLE OF CONTENTS

CHAPTER

1 INTRODUCTION 1

2 ILL-FORMEDNESS AND THE CASE FOR PRAGMATICS 3

2.1 Ill-Formedness, Redundancy, and Pragmatic Context 3
2.2 The Importance of Pragmatic Context 5
2.3 Classes of Ill-Formedness in Terms of Available Constraints 8
2.4 Alias Errors and Motivating Examples 11
2.5 Research Goals and Focus 14

3 EXISTING APPROACHES FOR HANDLING ILL-FORMEDNESS 17

3.1 Syntactically Driven Methods 17
3.2 Semantically Driven Methods 19

4 OVERVIEW OF A PRAGMATICS-BASED APPROACH 23

4.1 Basic Approach: Linking to the Pragmatic Context 23
4.2 Comparing the Linking Approach to Alternative Approaches 26
4.3 Selecting the Level for Linking 28
4.4 Wildcarding for Syntactic and Semantic Constraints 29
4.5 The Expert Advising Setting 31

5 DOMAIN PLAN TREES FOR MODELING PRAGMATIC CONTEXT 33

5.1 Role of the Domain Plan Tree in the Model 33
5.2 Existing Approaches to Plan-Based Models of Pragmatic Context 33
5.3 Theoretical Description of Domain Plans 36

5.3.1 Actions and Ground Plans 36
5.3.2 Plan Schemata 37
5.3.3 Plan Schemata as Subactions in Larger Plans 38

5.4 Classifying Schemata for Modeling Plan-Building 39

5.4.1 Plan Classes 40
5.4.2 Plan Class Trees 41

5.5 Plan Class Trees as a Plan-Building Model 44

6 METAPLANS TO MODEL THE PROBLEM-SOLVING CONTEXT 47

6.1 The Problem-Solving Context in the Expert Advising Setting 47
6.2 Existing Models of the Problem-Solving Context 49

6.2.1 Working Directly from the Domain Plan Tree 49
6.2.2 Discourse Models for Problem-Solving Dialogue 50

6.2.3 Uses of Metaplans in Pragmatic Modeling 51

6.3 Overview of a Metaplan Model 53

6.3.1 The Metaplan Context Tree 53
6.3.2 Four Classes of Metaplans 54

6.4 The Plan-Building Metaplans 55

6.4.1 Metaplans that Outline the Domain Tree 56

6.4.1.1 Build-Plan 56
6.4.1.2 Build-Subplan 58
6.4.1.3 Build-Subaction 58

6.4.2 Metaplans that Fill Variables 59

6.4.2.1 Instantiate-Var 59
6.4.2.2 Constrain-Var Plans in General 61
6.4.2.3 Add-Boolean-Constraint 63
6.4.2.4 Add-Scalar-Constraint 63

6.5 Query Metaplans 64

6.5.1 Plan Feasibility Queries 64

6.5.1.1 Ask-Pred-Value 65
6.5.1.2 Check-Pred-Value 65

6.5.2 Slot Data Queries 66

6.5.2.1 Ask-Existence 66
6.5.2.2 Ask-Cardinality 67
6.5.2.3 Check-Cardinality 68
6.5.2.4 Ask-Fillers 68
6.5.2.5 Limit-Cardinality 69
6.5.2.6 Sort-Set-by-Pred 70
6.5.2.7 Ask-Attribute-Values 70

6.6 Evaluative Metaplans 71

6.6.1 Evaluate-Plan, Evaluate-Subplan, and Evaluate-Subaction 73
6.6.2 Higher-Level Evaluative Metaplans 74

6.7 Informing Metaplans 75

6.7.1 Inform-Goal 78
6.7.2 Inform-Constraint 79

6.8 The Combined Metaplan and Domain-Plan Model 80

7 LINKING TO PRAGMATIC CONTEXT TO RESOLVE ILL-FORMEDNESS 83

7.1 A Framework for Heuristics in the Metaplan Tree 84
7.2 Probable Query Prediction: Heuristics from the Context Itself 85

7.2.1 Heuristic Use of Tree Shape and Problem-Solving Patterns 86

7.2.1.1 Tree Distance as a Model of Coherence 86

7.2.1.2 The Typical Problem-Solving Pattern 88
7.2.1.3 Representing Alternate Problem-Solving Patterns in the Tree 90

7.2.2 Predicting Plan Tree Growth from Agent's World Knowledge 93

7.2.2.1 Agent World Knowledge vs. Plan Knowledge 94
7.2.2.2 World Knowledge Relevant to the Planning Model 96
7.2.2.3 World Knowledge Not Requiring Deduction 97
7.2.2.4 World Knowledge Available in the Setting 98
7.2.2.5 The Agent World Knowledge Model 98
7.2.2.6 Effects of Agent's World Knowledge on Subplan Growth 100
7.2.2.7 Effects of Agent's World Knowledge on Query Metaplans 102
7.2.2.8 Effects of Agent's World Knowledge on Variable Instantiation Metaplans 103
7.2.2.9 Using Agent World Knowledge Established During Consultation 105

7.2.3 Predicting Domain Plan Choices Statistically 107

7.3 Heuristic Control from the Query Parse 109
7.4 Linking to the Partial Interpretations of Ill-Formed Queries 111

8 PRAGMA IMPLEMENTATION DESCRIPTION 115

8.1 Introduction 115
8.2 Wildcard Parsing 115
8.3 Atomic Sentence Set Logical Representation 117
8.4 Plan and Metaplan Implementation 120
8.5 Domain Plan Library, Database, and Type System 123
8.6 Building the Metaplan Tree 124
8.7 Heuristic Component Implementation 125
8.8 Testing for Links 126
8.9 Integration with the Janus NL System 126

9 RESULTS 129

9.1 Introduction 129
9.2 Presentation of Example Results 129

9.2.1 Damaged Vessel Domain Examples 130

9.2.1.1 The Initial Context 130
9.2.1.2 Example D1: What is the VOCATION of Fox? 132
9.2.1.3 Example D2: What is the location of FIX? 136
9.2.1.4 Example D3: Is Fox in the RED? 138
9.2.1.5 Example D4: Does Thorn have on board a SHARE ER-211 relay? 139
9.2.1.6 Example D5: How many BRUISERS are C1? 141

9.2.2 SPA Domain Examples 143

9.2.2.1 Initial Context Description 143
9.2.2.2 Example S1: What is the HEADINESS of Assertive? 147
9.2.2.3 Example S2: What is the detection probability for SPA I? 149

9.2.2.4 Example S3: What is the SAW readiness of Assurance? ... 150
9.2.2.5 Example S4: Which TAGS are assigned to SPA 2? ... 151

9.3 Factors that Affect Example Difficulty ... 152

9.3.1 Inherent Ambiguity of the Wildcard Word ... 152
9.3.2 Degree of Contextual Constraint ... 154
9.3.3 Logical Step Size ... 155

9.4 A Theoretical Issue: Modeling Alternate Perspectives ... 156
9.5 System Design Lessons ... 158

9.5.1 Dependent Variable Instantiation from the Database ... 158
9.5.2 Implications for Heuristic Search Control ... 160

9.6 Summary ... 161

10 CONCLUSIONS ... 163

10.1 Summary ... 163
10.2 Implications of this Approach ... 164
10.3 Areas for Further Work ... 166

BIBLIOGRAPHY ... 169

LIST OF FIGURES

4.1: Choosing a Level for Matching 28
4.2: Interpretation with Single Wildcard 30
4.3: Interpretation with Two Wildcards 31
5.1: Assertions Describe a Blocks World State 36
5.2: Example Primitive Action 36
5.3: Example Ground Plan 37
5.4: Example Plan Schema 37
5.5: Plan Schema with Preconditions 38
5.6: Schema with Multiple Arguments 38
5.7: Example Plan Class 40
5.8: Plan Class for Support-Bell 41
5.9: Plan Class for Support-Bell-Using-Bean 41
5.10: Plan Class Omitting Inherited Preconditions 42
5.11: Plan Class for Sail 42
5.12: Plan Subclass of Sail 42
5.13: Subclass Introducing Subactions and New Variable 43
6.1: Example Dialogue from Litman 52
6.2: Plan Class Header for Increase-Readiness 57
6.3: *Build-Plan* Node for Increase-Readiness 57
6.4: A Multiple-Assertion Constraint 63
6.5: Assertion Restricting Set to Five Closest Vessels 64
7.1: Default-Downward-Subplan H-Rule 86
7.2: Default-Downward-Subaction H-Rule 89
7.3: Evaluate-Plan-from-Build-Plan H-Rule 91
7.4: Evaluate-Plan-Sibling H-Rule 92
7.5: Eval-Plan-Alternate-Instantiation H-Rule 93
7.6: Block-Known-Failure H-Rule 101
7.7: Block-Known-Failure-Instantiate-Var H-Rule 101
7.8: Block-Query-of-Pred-Known-OK H-Rule 102
7.9: Reduce-Ask-Fillers-with-Large-Set H-Rule 103
7.10: Boost-Constrain-Var-with-Large-Set H-Rule 104
8.1: WML for *List the readiness of the Fox* 117
8.2: WML for *What is the readiness of the Fox* 118
8.3: Representation of *List the C1 ships in the Indian Ocean* 120
8.4: Representation of *Which ships have visited each port* 120
8.5: The **Restore-Slot-Readiness** Plan Class 121
8.6: The **Replace-Ship** Plan Class 121
8.7: The ***Build-Subplan*** PS-Plan Class 122
9.1: Input Form for *What class is Sterett?* 130
9.2: Partial Tree from Matching *What class is Sterett?* 131
9.3: Input for Example D1 133
9.4: Partial Tree for Example D1 134

9.5:	Three Links from Example D1	135
9.6:	Full List of Links for Example D1	135
9.7:	Input for Example D2	136
9.8:	Partial Tree for Example D2	137
9.9:	Input for Example D3	138
9.10:	Input for Example D4	139
9.11:	Partial Tree for Example D4	140
9.12:	Matching Assertions for Example D4	140
9.13:	Input for Example D5	141
9.14:	Partial Tree for Example D5	142
9.15:	Context Query in SPA Domain	145
9.16:	Partial Tree for SPA Context Query	146
9.17:	Input for Example S1	147
9.18:	Partial Tree for Example S1 with Cutoff 50	147
9.19:	Additional Partial Tree for Example S1	148
9.20:	Input for Example S2	149
9.21:	Input for Example S3	150
9.22:	Input for Example S4	151
9.23:	Partial Tree for Example S4	151
9.24:	Summary Results with Wildcard Part of Speech	153

ACKNOWLEDGMENTS

I am deeply grateful to Ralph Weischedel for his help, direction, inspiration, and support during the course of this research, both at Delaware and more recently here at Bolt Beranek and Newman. Through it all, he deftly balanced the roles of advisor, manager, colleague, and friend.

I would also like to thank the other members of my committee, and especially Dan Chester, who more than filled in at Delaware after Ralph's departure with technical advice and helpful organization. My indebtedness to Sandra Carberry is evident throughout this dissertation.

My personal thanks, also, to Gail Wine and Lyle Ramshaw for crucial support of many different kinds, especially during these final months.

This work was supported in part by the National Science Foundation under grants IST-8311400 and IST-8419162 and by the Defense Advanced Research Projects Administration under Contract No. N00014-85-C-0016; their support is gratefully acknowledged. The views and conclusions contained in this document, however, are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

CHAPTER 1

INTRODUCTION

Many examples of ill-formedness that people routinely and easily correct can be resolved by a natural language system only if it makes use of knowledge of the pragmatic context. This investigation centers around examples of *alias* errors, where the ill-formedness is due to a single word that is incorrect but still lexically understood, as with the substitution of *on* for *in* in the phrase *stay on good shape*. Localizing and resolving such errors frequently depends on pragmatic knowledge. This thesis presents a model for pragmatic context within expert advising dialogues, where an agent who is building a plan to solve a problem consults with a domain expert, and develops methods for applying that model to resolving ill-formed input.

Metaplans are used to model the structure of the agent's problem-solving behavior, both the gradual refinement of the domain plans being considered and the connection between them and the queries motivated by them. The partially-specified domain plans that the agent is considering are represented by nodes in a plan classification hierarchy, and these classes of domain plans in turn serve as arguments to the problem-solving metaplans. The expansion and search of the metaplan tree that models the problem-solving context is governed by heuristics based both on its metaplan structure and on a model of the agent's world knowledge. This model can be used to track the problem-solving moves implicit in a sequence of well-formed queries and also to predict likely moves and queries as determined by the context which can then be linked to the partial interpretation of an ill-formed query suggesting corrections for the ill-formedness.

The substance of this approach has been implemented in a system called Pragma, although some elements of the metaplans and heuristics have been developed further in the theoretical presentation that the implementation currently supports. Pragma suggests corrections based on pragmatic context for alias errors in naval domain queries, and it has been successfully demonstrated as an adjuct to the Janus [57] natural language interface system. The techniques used could also be extended to other classes of ill-formedness and to generating cooperative responses. The system demonstrates that a model capturing the pragmatic structure of a particular discourse setting can be used to increase the robustness of a natural language interface.

Chapter 2 begins by discussing the importance of pragmatic context in resolving ill-formedness and characterizing the class of alias error examples. Chapter 3 considers existing approaches to ill-formedness, while Chapter 4 gives an overview of the pragmatics-based approach taken here. The classification method for representing partially-specified domain plans is presented in Chapter 5, while Chapter 6 covers the metaplans used to model the plan-building context and Chapter 7 deals with the heuristics used to direct exploration of the tree and to rank alternate possible solutions. Chapter 8 then describes the Pragma implementation, Chapter 9 presents and analyzes

the results of the test runs performed, and Chapter 10 summarizes and discusses areas for further work.

CHAPTER 2

ILL-FORMEDNESS AND THE CASE FOR PRAGMATICS

2.1 Ill-Formedness, Redundancy, and Pragmatic Context

Natural language [NL] often achieves its communicative purpose in spite of significant amounts of ill-formedness. Even when a piece of text contains misspellings or incorrect words or does not follow the rules of grammar, it is often possible for an understanding system to interpret it correctly if the system can make effective use of the various sorts of contextual knowledge that are available in the communicative situation. This is possible because each item in an NL utterance exists in a web of many layers of constraints, including the constraints that hearers depend on in recognizing words from their constituent sounds, parsing the syntactic form of the sentence, deriving a semantic interpretation, and fitting that interpretation into the pragmatic context. That web of constraints is rich enough in meaning that many sorts of ill-formedness can be automatically detected and corrected if the information in the constraints can be effectively used. After discussing ill-formedness generally and existing techniques for handling it, this thesis presents new mechanisms for making use of pragmatic information in interpreting ill-formed input.

A practical motivation for computer-based language processors to be able to handle ill-formedness is that it occurs quite frequently in actual inputs to computer systems. One study has shown that as much as 25% of the input to a computer system can be ill-formed [16]. Systems that can make use of context to detect and correct for such errors will be much more useful than systems that simply ignore or reject input that is not exactly correct. Human communication using natural language, it often turns out, is efficient not because ill-formedness does not occur but because the contextual and other constraints are so often strong enough that the hearer can correct automatically for the ill-formedness, without either being misled or requiring explicit clarification dialogue. Indeed, any claim to be a "natural" language processor requires some ability to handle ill-formed input intelligently.

Exactly what counts as well-formed or ill-formed depends in part on the grammatical standards being used by the author and the interpreting system. Split infinitives, for instance, are ill-formed in some grammars, but well-formed in others. This lack of a universal standard makes it hard to give a formal definition of ill-formedness, unless one invokes some implicit standard like the opinion of "the typical listener" [56]. The exact boundaries of ill-formedness, however, are less important to us here than techniques for dealing with it. We can be sure that any natural language communication system will classify various kinds of input as ill-formed because the robustness demands of the communication process require a grammar that includes enough redundancy to detect and correct a large fraction of the errors produced.

The ability to recognize, localize, and correct ill-formedness is dependent on the amount of redundancy that exists for each utterance in its context. A redundant form of communication is more verbose than a minimal-length encoding of the information that needs to be transmitted, and it is those extra bits that allow many errors to be recognized and corrected. Such redundancy is found at every level in NL communication. There is redundancy at the lexical level because the area of legal words does not densely fill the space of possible letter combinations. Thus a typographical error from an actual word is unlikely to be another legal word; it often instead is a meaningless letter sequence that triggers awareness of the error, and allows searching for "nearby" sequences that are legal and appropriate in the context.

Grammatical rules like agreement rules often add a different kind of redundancy to an utterance. There are some cases where the agreement information is needed to resolve a grammatical ambiguity, for example,

Do you know the insurance payments on the car that were/was *totalled yesterday?*

But such examples are not typical, so that an agreement error is unlikely to produce a legal sentence with an alternate interpretation; it usually produces an ill-formed sentence which the reader can easily correct, since the grammatical roles are still clear:

My opinions, in spite of what you may say, is no business of yours.

Thus, the bits devoted to inflecting verbs to agree with their subjects are often redundant, and contribute to robustness.

Even erroneous sentences that satisfy the grammatical rules may still be recognized as anomalous if they contradict the semantic constraints that define which syntactic combinations produce meaningful interpretations, as in examples like this,

The back windows of my idea are in remission.

The fact that some syntactically correct strings are still semantically ill-formed shows that independent redundancy is introduced at the semantic level. Any sublanguage that was fully non-redundant (like telephone numbers, that can be any seven-digit number) would provide no internal opportunity at all for detecting such errors; the only way to detect an incorrect value in that case is to have some external source of information and redundancy.

In NL research, knowledge about the situational context of an utterance is termed "pragmatic" knowledge, and it provides an additional layer of constraints on well-formedness. There are examples in which pragmatics provides the only clue to the ill-formedness. If a woman has two daughters, the younger of which lives in Massachusetts and the older in California, and a guest asks

How is your younger daughter doing?

and the woman replies with

She has a new apartment in LA and is working in the movies.

the mistake about which daughter is meant makes the answer pragmatically ill-formed in that it does not answer the guest's question, and the guest will probably try to correct

the error. (Not all *non sequiturs* are ill-formed, of course. Speakers sometimes deliberately flout the Griceian principle of relevance [21] on the topic level in order to communicate at the discourse level, perhaps implying that they would like to change the subject, but that explanation seems unlikely in this case.) There are also many cases of ill-formedness in which non-pragmatic constraints are enough to identify and localize the error, but pragmatic information is still essential to resolving it. If a student asked

Is there room in CIM 360?

lexical knowledge is enough to say that *CIM* is not a legal word in this context, but not enough to say whether *COM* (Communication) or *CIS* (Computer and Information Sciences) was actually intended. However, the pragmatic context might well be able to resolve this, for example if the student were known to be a CIS major who had just been asking about her major requirements.

At each level of human language processing, therefore, there are constraints that introduce the redundancy that is needed to recognize and often resolve ill-formedness. If NL systems are to achieve the same degree of robust understanding, they also must learn to exploit this full web of constraints on every level. It is true that while computer methods for NL understanding historically have considered only well-formed input, there has been a strong current of research lately on techniques to allow parsers to diagnose and correct ill-formedness. However, the thrust of this thesis is to argue that existing mechanisms for dealing with ill-formedness have not made adequate use of pragmatic context, which, while hard to formulate, is actually the most powerful source of constraints for identification and resolution of ill-formedness. Once a dialogue context has been established, the set of utterances that would be coherent responses in that given situation is far smaller than the set of all meaningful utterances, and it is the power of the pragmatic constraints that achieve that reduction which must be made available to the NL system. In this thesis, we suggest a way of encoding that pragmatic context in the particular domain of expert advising systems, a method that builds on research in plan recognition and plan modeling, and then a way of using that pragmatic context model for the resolution of ill-formed input, including methods for combining that pragmatic knowledge with lexical, syntactic, and semantic constraints.

2.2 The Importance of Pragmatic Context

While each of the levels of processing generates its own set of constraints to filter the set of well-formed utterances, there is a special character to the pragmatic constraints, as will be pointed out in this section, that makes them particularly important in the process of working out the meaning of an ill-formed input. While the other levels of constraints can be relaxed, if necessary, in order to find an interpretation, pragmatic constraints cannot be relaxed without giving up on the listener's overall goal of assigning the utterance a role in the discourse context. This special importance of pragmatic constraints supports our contention that a pragmatic context model is a crucial resource in handling ill-formed input.

Since an utterance is recognized as ill-formed when it transgresses one of the constraints imposed by the interpretation rules of the language, whether syntactic, semantic, or pragmatic, one way to describe the effort to correct the ill-formedness is

as a search for the most closely related utterance that does meet the constraints, where the calculation of "closely related" includes whatever factors the system judges may help to explain the occurrence of the ill-formedness. Such factors include euphony, like the child complaining of having a *night bear*, close semantic relationship, like a parent confusing two of her children and saying *Mary* for *Joan*, or confusing syntactic structure, like subject verb agreement examples where the singularity of the subject is masked by a plural appositive:

> *The new system, both the files from Penn and the interface utilities from Delaware, were installed yesterday.*

The most closely related well-formed sentence is taken to be the intended meaning. Of course while the hearer usually proceeds on the assumption that the speaker is working with full knowledge of the language constraints, there are also cases like

> *They was sent home.*

where the reason for the ill-formedness may be a subdialect difference in the constraints applied by the hearer and speaker. In special settings, also, completely non-linguistic criteria can play a role; in interpreting typed text, for example, missing or extra letters may be attributed to physical performance errors.

Many ill-formedness researchers have phrased this search in another way; rather than speaking from the generation perspective of the factors that may have produced the ill-formedness, they refer instead to searching within the system's constraints for which language rules to relax so that the ill-formed sentence will be well-formed in the larger grammar recognized by the relaxed rules. (See Section 3.1 for references.) The same kind of incremental search is implied, but here phrased in terms of a sequence of rule relaxations to be explored, which is the analogue here to the explanatory factors for ill-formedness in the previous view. Some researchers have thus proposed a set of relaxed rules with attached scores, so that the search for a corrected interpretation would proceed by testing relaxed rules in the order of increasing seriousness until an interpretation was found.

While the relaxation perspective seems less intuitive, it has the advantage that it does not need to hypothesize an explicit well-formed utterance from which the ill-formed one can be derived. For example, when Weischedel and Sondheimer [56] propose treating the utterance

> *My car drinks gasoline.*

by relaxing the semantic constraints on subjects of **drinking-events**, they can understand the point of the phrase without modeling a specific alternate wording.[1]

One crucial point is missed, however, in these formulations of the search for corrections for ill-formedness, and that is the role of pragmatic context in the search. If we think of a search for "closely related" utterances, the relationship that must be the focus is not that between the ill-formed sentence and the hypothesized correct one, but

[1]While this example is not strictly ill-formed, many researchers propose handling such cases by the same mechanisms as ill-formed ones.

between the ill-formed sentence and the context into which the hearer must fit it in order to understand it. For example, suppose a mail system user stated

Messages 5-9 are all about the reorganization.

and then said

Forward those messages for Smith.

The query as it stands is semantically ill-formed, since there is no interpretation of *for Smith* modifying the *forward* command, and if *messages for Smith* is taken together, the command then lacks the required destination specification. We can easily see two related queries, however, that would be well-formed in the correct contexts, one where *for* is replaced by *from*, if there were a set of messages from Smith and a default destination implied by the context, and another where *for* is taken as *to*, so that *Smith* becomes the destination for *those messages*. Even though the metric for likely errors may predict that *for* and *from* are more closely related because of euphony than *for* and *to*, the pragmatic context in this case means that only the latter interpretation can be made to fit. Thus, the search for a related interpretation does not begin with the ill-formed sentence, but with the pragmatic context, which defines the set of utterances that can be coherent additions to that known setting.

Alternatively, when the search for a resolution to the ill-formedness is conceived as a search for a linguistic constraint to relax in order to understand the input, we find that pragmatic constraints play the same special role. Because pragmatic constraints deal with the role the utterance as a whole will play in the discourse, they cannot be relaxed in the same way as a lexical, syntactic, or semantic constraint without making it impossible to discern the intent of the utterance. Even if an utterance is otherwise completely well-formed, if the pragmatics is off, the hearer must struggle to find another interpretation that does fit, and that continues to be true even if other constraints must be stretched to make the alternate interpretation possible. For example, suppose a student who had just asked

Is there space in CIS 630?

and had been told that there was not then proceeded to ask:

What about CIS 160?

We would expect the adviser to protest the pragmatic incongruity of that follow-on query, perhaps by asking

Do you mean 610?

The pragmatic context from the first query sets up strong expectations about the sort of goal the speaker is pursuing and about the speaker's status as a graduate student, and the second query, although in itself perfectly well-formed, clashes so severely with those expectations that the hearer is forced to hypothesize some unnoticed ill-formedness or error in interpretation rather than settle for the lack of pragmatic coherence that the obvious interpretation entails. On the other hand, if an utterance can be made to match the pragmatic context, even gross violations of other constraints can be easily repaired or overlooked. Thus if the same student had followed that first query with one of the following:

What on 640?
Is space there 640?
Spaces 640?

we would expect the hearer to be able to resolve the ill-formedness and respond directly.

Of course, the pragmatic context does not always produce as powerful a set of expectations as in this example. In some contexts, the variety of follow-on utterances that would be pragmatically well-formed is much greater. An unsignalled shift in the speaker's goals may be the reason behind apparent incoherence, for example, if the student's second query was on behalf of a friend. There are also cases where a reinterpretation of the pragmatic context establishes a coherence that seemed to be lacking; the given example might become coherent if the student turned out to be writing a story about overcrowded classrooms. Still, while the pragmatic context may need to be revised, it cannot be relaxed, in the sense that a syntactic rule can be relaxed.

It is in this sense that the pragmatic context forms a boundary condition for the interpretation problem, a given that must be dealt with as it stands, since to understand the utterance means exactly to connect it in a coherent way with its pragmatic context. While other constraints can be stretched in the search for an interpretation, giving up on the constraints of pragmatic context would mean abandoning the goal of coherence itself. Thus we can expect that a model of pragmatic context will provide a powerful source of constraints to direct the search for corrections for ill-formedness.

The use of pragmatic constraints for this purpose has been limited historically by the greater difficulty in formulating adequate models of pragmatic context. NL research has usually focused first on isolated sentences, at which level only lexical, syntactic, and semantic constraints can be applied, since the pragmatic context is unspecified. However, recent work in plan recognition, as pointed out in Section 5.2, has made available a technique for pragmatic modeling that allows us to begin to explore the application of pragmatic constraints in a setting that badly needs them.

2.3 Classes of Ill-Formedness in Terms of Available Constraints

In this section, we consider ways of classifying examples of ill-formedness with a view to selecting a useful class of examples for our exploration of the utility of pragmatic context. In this survey of types of ill-formedness, we examined corpora of naturally-occurring dialogue from various contexts including a computer interface in a ship loading task, students consulting an academic adviser, and a radio talk show for financial advice. Many ill-formed examples were identified, for which various classification schemes are possible. The most useful classification seems to be one based on the levels of constraints that are sensitive to that particular kind of ill-formedness.

Classifying examples of ill-formedness is an interesting problem in itself, and NL researchers working with ill-formedness have used various approaches. The most

common approach is to classify them based on either the grammatical constraint being violated or on the assumed etiology of the error, resulting in categories like agreement errors or spelling errors. Such classifications are at best approximate, since many errors fit more than one classification. For instance, any agreement error could also be seen as a spelling error or incorrect word example. This kind of classification requires both a judgment about what utterance was actually intended and also a heuristic choice about how to categorize the difference between the actual and intended utterances. (Naturally, there will be some ill-formed examples that are so garbled that the hearer is unable to make any connection between them and the context or any guess as to the speaker's actual intent, and such examples will simply need to be lumped together at the bottom of the ratings.) We suggest instead a classification based on the levels of constraints applied by the NL system in interpreting the utterance and the results of the ill-formedness on those different levels of constraints.

We can approach this constraint-based classification by first considering well-formed utterances, and noting that the purpose of the constraints in an NL system is to map out the legal derivations from intended meanings to realized utterances. The understanding system applies those constraints in reverse to the utterance to deduce from it the possible original meanings. An unambiguous utterance is one where, at least in context, this interpretation process produces a single meaning that is thus established as the intent behind the utterance. In the great majority of discourse contexts, such an unambiguous utterance is the natural strategy for a speaker to adopt to communicate her message. If we assume that the speaker's and hearer's context and language models agree, such an utterance will be decoded in a way that conveys its intended message.

An ambiguous utterance that allows for more than one interpretation in context presents the danger that the meaning discerned by the hearer may not be the one intended by the speaker. Such ambiguity may be unintentional, where the speaker is not aware of the alternate interpretation, in which case miscommunication may well result, and the utterance is, in a sense, ill-formed for its communicative purpose. This is different, of course, from cases where intentional ambiguity is used to serve a particular purpose in the discourse, as is true of puns where the hearer is expected to recognize and enjoy the ambiguity, or of cases where ambiguity is used to avoid communicating more than necessary.

One broad definition of ill-formedness would include all utterances that do not achieve the communicative purpose of their speaker, making this effectively equivalent to miscommunication. That would include cases where the communication failure was due to differences in context as perceived by speaker and hearer. A problem with that definition, of course, is that the hearer may have no way even in principle of detecting that sort of ill-formedness. Another problem is that this definition rules out examples like minor misspellings in written text that do contradict a linguistic constraint but in such a way that the communicative purpose of the utterance is not threatened.

When we speak of ill-formedness, we are instead focusing on discernible cases of miscommunication, examples that will be recognizable to the hearer as ill-formed, while miscommunications that result in a single incorrect interpretation on the part of the hearer will not in general be recognizable as ill-formed, an example, again,

being a mistake in reporting a telephone number. Of course, the most frequent way in which miscommunication becomes obvious to the hearer is through the failure of the utterance to result in even a single interpretation through failing to meet some constraint of the language. There are also rarer examples where the existence of multiple interpretations reveals the presence of ill-formedness, as in the following example, where *him* was a typo for the intended *Jim*:

Speaker 1: *I could take Mary, Mark, or Jim.*
Speaker 2: *Take him.*

In this case the ill-formedness is due to the failure of the pronoun reference to be restricted enough to select a single referent. Thus ill-formedness, whether it results in zero or multiple interpretations, is detectable miscommunication revealed to the hearer by the violation of some linguistic constraint.

Naturally, this definition does make ill-formedness dependent on the hearer's model of those linguistic constraints and of the context. If that model is off, there will be examples where the hearer will detect apparent constraint failures in utterances that are actually well-formed, or not detect them in ill-formed examples. An example of the former would be when the hearer does not know a figure of speech like *kick the bucket*, and so fails to find an interpretation for a sentence containing it; the latter can be seen in the utterance

Leslie left her books at home today.

when the hearer is not aware that the Leslie in question is male. This sort of misdiagnosis is, of course, more of a problem with NL systems at this point than with human hearers, since current systems often contain significant omissions and flaws in their representations of linguistic and pragmatic constraints.

Given this definition, it seems that the most useful classification for types of ill-formedness is to group them according to the class of constraints that are violated, whether those constraints are lexical, syntactic, semantic, or pragmatic. Misspelled words would be examples of lexical ill-formedness. Syntactic ill-formedness includes specifically syntactic tests like agreement failures along with cases where an incorrect or missing word leaves a sentence with no successful syntactic interpretation. Semantic ill-formedness arises, for example, from incorrect words or prepositions, where there may be a successful syntactic interpretation, but one that does not have any corresponding semantic reading. Pragmatic ill-formedness, in turn, covers cases where there is even a valid semantic reading, but that reading cannot be fit into the current pragmatic context.

The advantage of this classification scheme is that it characterizes the ill-formedness in terms of the type of knowledge needed to at least localize the error, so that an example of syntactic ill-formedness can be localized on the basis of syntactic constraints. That is different, of course, than saying that the same kind of constraints will be sufficient to resolve the error. Quite the contrary, it is very often true that even a lexically localizable error may require pragmatic information to resolve, if the hearer in fact can resolve it at all. The level of knowledge required to resolve the error is usually much greater and also much more dependent on the circumstances in which it occurs than that required to identify it, and thus more difficult to use in a classification

scheme. Also, while this classification is based on different levels of constraints, that does not imply any commitment to a sequential processing model that begins with lexical processing and works up. Indeed, although processing order is not the point here, the usefulness of pragmatic constraints established in this work might well argue for bringing them to bear before the final stages of processing.

2.4 Alias Errors and Motivating Examples

The choice of examples can have an important bearing on an investigation of this sort, since they will determine to some extent the kinds of mechanisms that will be appropriate. In order to explore the use of pragmatic knowledge in resolving ill-formedness, we are interested in a class of examples

- that frequently can be resolved only by resorting to pragmatic information, and
- that are not hard to generate.

The primary criterion in choosing a class of examples is to select one for which syntactic and semantic techniques alone are usually not adequate. Since our hypothesis is that pragmatic constraints are the most useful ones in resolving ill-formedness, it provides a stronger test if the examples on which our techniques are tested are ones that are too hard for methods that do not use pragmatic knowledge. If we can show that pragmatics allows us to resolve even that sort of difficult class, it will then be easy to see how the addition of pragmatic guidance to techniques already demonstrated for classes of ill-formedness that are easier to resolve will also be able to improve their performance.

An additional practical restriction is that examples of the given class not be too hard to generate. This comes simply from the need to be able to generate examples within the domain for which a plan library and database of facts are available to support the pragmatic modeling. While it would be preferable to work solely from examples gleaned from naturally occurring dialogue, it is not feasible to collect sufficient examples in a single constrained domain. The search for an example set of reasonable size requires resorting to manufactured examples, and thus argues for a class of examples that can be generated fairly easily.

The class of examples that we propose to work with are single-word *alias errors*. By *alias errors*, we mean examples of semantic or pragmatic ill-formedness whose etiological explanation involves an error on the lexical or syntactic level. For example, an error like a typographical transposition normally produces an ill-formedness identifiable by lexical constraints, since the letter combination produced is not a legal word. On occasion, however, the new sequence turns out by chance to be another word, in that sense an *alias*, which satisfies lexical and perhaps also syntactic constraints, so that the ill-formedness is not identified until it causes a semantic or pragmatic failure. Such errors thus happen to evade at least some levels of the redundancy that usually catches random permutations of a piece of natural language text. A popular type of alias error is spelling mistakes that happen to be also legal words. If *in* is misspelled as *jn*, that can be caught by a lexical processor, but if it is misspelled as *on*, as in the statement

You need exercise to stay on *good shape.*

a higher-level constraint is needed to catch the error, if, indeed, it can be caught at all. Another example would be syntactic alias errors when a grammatical error produces a sentence with a partially-successful alternate reading, as in the following example (ill-formed unless the speaker is a senator):

I lied down on the floor.

Alias errors are interesting and particularly difficult to resolve both because they frequently have multiple possible corrections and because the locus of the error itself is obscure. As an example of the former, the *exercise* example above could be corrected either to

You need exercise to stay in *good shape.*

or to

You need exercise to stay on good terms.

For the latter, consider the following naval domain query:

Will the detection of the Swordfish be followed by the other subs?

It does not make sense in this domain for vessels to follow events, but it is not clear where in the query the problem lies. Considering only single-word errors, the speaker might be wondering about *the direction of the Swordfish* in comparison to the course of the other subs, or if its detection would be *noticed* by the other subs, or if its detection would be followed by the other *tasks*. Some other classes of ill-formedness like agreement errors share the former property of having multiple possible corrections, so that

Is the subs in port?

could be corrected either to *is the sub* or *are the subs*, but alias errors are unique in terms of this difficulty in localizing the problem. The fact that the error results in an alias for a form that is at least partially coherent can mislead the hearer as to its actual source.

This combination of difficulties has made alias errors particularly difficult to handle. For example, to handle the misspelling of *on* for *in* by a direct extension of the usual approach to spelling errors would require treating each word in the sentence as perhaps misspelled. This is, in fact, the approach taken by Trawick [53], who defines a fixed "neighborhood" distance between two words in terms of the number of letters changed, and then tries to parse the sentence with each possible substitution for each word. The combinatorics of this method make it impractical for general use. As Granger [19] correctly argues, semantics and pragmatics usually provide a stronger constraint on the possible fillers than lexical closeness, so that it is more efficient for the recovery strategy to make use first of those contextual constraints, rather than to proceed by blindly searching the dictionary for respellings of each word. It is exactly the difficult cases like alias errors that make clear the importance of bringing pragmatic knowledge to bear.

Note that our definition of alias errors rules out the related but even more

problematic class of examples where the low-level error creates an alias that successfully evades not only lexical and syntactic but also semantic and pragmatic constraints. An undetectable error is, in a sense, a perfect alias, because the hearer, even applying context, has no way of discovering it. Such errors may be caught only much later, when the world model built up from them proves to conflict somehow with the real world, or they may function as disinformation that is never caught. Clearly, we are only interested in examples that can be identified as ill-formed given the relevant context.

Within the class of detectable alias errors, we can also distinguish those that are *correctable* either with or without using pragmatic information. Roughly ranked from easy to hard, we thus have the following classes:

1. Detectable and correctable without using pragmatics:
   My neighbor's tabby cot is good with children.

2. Detectable without using pragmatics, correctable with pragmatics:
   I'm a computer major. Is there room in CIV 360?
   (Where there is no Civil Engineering course numbered 360.)

3. Detectable without using pragmatics, but not correctable at all:
   My friend wants a new caw for Christmas.
   (*Car*, *cat*, or *cap* would all fit.)

4. Detectable and correctable using pragmatics:
   It's cold up there. You'll need to get a heavier goat.

5. Detectable using pragmatics, but not correctable at all:
   The two ships were 50,000 miles apart.
   (Could be *50*, or *feet*, *yards*, or *meters*.)

6. Not detectable even using pragmatics:
   They're excellent fish. I caught them in Portsmouth.
   (Where *bought* was intended.)

Here we are primarily concerned with the classes 2 and 4, those that can be corrected using pragmatics, whether or not the presence of the error can be recognized without pragmatics. Class 1 examples, while still alias errors, are amenable to purely semantic methods, and classes 3, 5, and 6 are examples where either the pragmatic context is insufficiently constrained or the alias is sufficiently devious that no correction at all is possible.

Within the larger class of alias errors, we will focus on those that are caused by a single-word lexical error. That sort of error is easier to generate, and still demonstrates all the important features of the class, including arbitrarily complex ambiguities of sentence interpretation. There are more complex errors that can still be corrected, given a strong enough pragmatic context, but we believe that single-word errors cover a usefully large portion of the landscape. Certainly, any scheme powerful enough to handle single-word alias errors could also handle non-alias misspellings and many other classes of ill-formedness like agreement failures, although weaker, faster, more specific methods can also handle some of them. Thus it seems that the mechanisms for modeling and applying pragmatic context that are worked out for single-word errors will also generalize to broader classes of ill-formedness.

One possibly misleading feature of these single-word alias errors is that our generated examples, in order to suggest how they might have arisen, are usually close typographical neighbors of the suggested intended forms. Our approach to correcting the errors, however, as will be seen, ignores that lexical similarity, so that the word *vocation* might find the three suggested corrections *speed*, *class*, and *location*, without any preference yet assigned to the latter because of its lexical similarity. It might seem foolish to suggest the use of complex techniques like pragmatic models when simple information like typographical closeness is being ignored. However, while lexical information might help with these examples, there are many cases of alias errors reflecting incorrect word choice where it would not help. These examples are used because they are easy to follow, but the techniques for applying pragmatic knowledge which are the focus here are kept free of dependence on lexical information because there will be many cases where that information would not be helpful and where pragmatic context offers the only clues.

2.5 Research Goals and Focus

Our overall goal is to explore the use of pragmatics for resolving ill-formedness, and particularly alias errors. Thus the primary focus is on developing mechanisms for modeling pragmatic context and for applying that model to suggest corrections for ill-formed inputs. The central theoretical issues involved in this effort include:

- what sort of pragmatic information is helpful for ill-formedness resolution,
- what kind of pragmatic model can capture the necessary features of the pragmatic context,
- how to predict from a given context in that model the space describing the agent's possible next moves and related possible queries,
- how the syntactic and semantic constraints in the ill-formed sentence can also be captured and used, and
- how a heuristic search can be organized that makes effective use of all these sources of information.

This theoretical core of the research is set in the context of an implementation of the developed approach within the Janus NL system, which serves as a testbed for the ideas, demonstrates feasibility, encourages clarity, and allows for experimentation. Full advantage is taken of the availability of Janus system components to help make the implementation more extensive and realistic than might otherwise be possible. However, there are still important secondary issues involved in a full integration of this system with Janus, such as an implementation in Janus of a partial parsing facility, that are beyond the scope of this effort.

In order to make significant progress on a topic of this scope, it is important to narrow the focus to a sensible subset of all the relevant issues. Thus, there are many issues that a full-fledged robust NL system would need to handle that are ignored here.

- We limit ourselves here to alias error examples in an expert advising

setting.

- We only consider examples where the system's plan library already contains all the domain plans relevant to the user's input.
- We also assume that the user shares with the system correct and complete beliefs about the possible domain plans that bear on the examples, ignoring cases where the user's query is based on incorrect plans.
- In our plan-based pragmatic model, we assume that the system will begin an interaction with a correct picture of the user's single top-level domain goal. Thus, while we do consider the use of our model in a tracking mode across a sequence of queries, it is always within the planning space dominated by the known top-level goal; we do not concern ourselves with the plan recognition issues involved in deducing the user's goals from her actions. As pointed out in Section 5.2, those issues are receiving a great deal of attention from other researchers, work that does not need to be duplicated here.
- No special provision is made for noticing interactions or interference between different actions of a plan.
- Our logical representation is a simple one, avoiding issues like representing time, complex quantification, or possible worlds.
- The database of facts used to describe the world is treated as fixed, taking advantage of the nature of the expert advising setting to assume that the user will not be performing actions during the consultation that would change the world's state, and that changes due to outside causes can also be ignored.
- We avoid cases where other issues like anaphora resolution would complicate the ill-formedness processing. For instance, in the sentence

  Chop the onion in a food processor and then FOO it.

  FOO might mean *add* or *fry* if *it* refers to the onion, but *empty* or *wash* if *it* refers to the processor. Such collisions would cause serious trouble for a general-purpose NL processor, though a well-founded pragmatic approach still offers the best path to a solution even of such difficult cases.
- We also do not explore here the generation component for interacting with the user about the detected ill-formedness and proposed corrections that any full implementation of pragmatics-based ill-formedness correction would also have to include.

Thus the research focus is exactly on formulating a model for pragmatic context, predicting from that model the space of the user's possible actions, and using those predictions in a heuristically sensitive way to identify possible corrections for alias errors.

CHAPTER 3

EXISTING APPROACHES FOR HANDLING ILL-FORMEDNESS

In this chapter, we survey the directions that earlier researchers have taken toward dealing with ill-formed input. (Surveys of other work in plan tracking and pragmatic modeling are included later in the appropriate chapters.) While many approaches have been suggested for how to encode and apply the additional knowledge needed for resolving ill-formedness, the attempts can be divided into two broad classes which we term *syntactically driven* and *semantically driven* and that correlate well with the approaches those researchers tend to use for parsing well-formed input. The former are approaches that are closely tied in with the syntactic parser and that rely on adjusting the input or the syntactic rules based on the presence of ill-formedness in the search for a complete parse, while the latter instead try first to derive the possible semantic structure of the input, using semantic constraints to suggest corrections for the ill-formedness, and perhaps then relating the correction to a full syntactic parse. Some of the semantically driven methods come close to using what we would class as pragmatic constraints in controlling the search for a resolution, but most of the methods discussed do not make any direct use of pragmatic knowledge, although a system that adopted one of these approaches could still apply pragmatic constraints after the fact to filter the corrected input for pragmatic well-formedness.

3.1 Syntactically Driven Methods

In a syntactically driven approach, some form of syntactic parse must be achieved first, a parse of some fragment, at least, if not of the whole sentence, before the system can draw on the semantics of the fragment in its semantic context to help resolve the problem. This forms an initial bottleneck for syntactic approaches, since at least a tentative decision needs to be made about what syntactic structure to try to build before semantic or pragmatic knowledge can be applied. This requires extending or relaxing the grammar in some way, so that it can handle the ill-formed input. The danger, of course, in thus loosening the grammatical constraints is that many spurious interpretations will also be accepted, or at least that much time will be wasted in eliminating the extra readings that syntax will now propose. Researchers in this tradition have focused on ways to control this relaxation process so that the best interpretation can be identified quickly.

The simplest style of extending a grammar calls for adding ill-formedness rules directly to the normal collection. This was the kind of ill-formedness processing available in the LUNAR grammar [60]. Another approach is to use a separate body of rules for ill-formed constructions, and to access the ill-formedness rules only after the sentence fails to parse normally. This style was used by Harris in the ROBOT system [25] for NL database access. He first tried to handle sentence fragments by adding extra arcs to the ATN used for well-formed input, but found that that slowed down the

processing of complete sentence input unacceptably, so instead created a second ATN for sentence fragments, that was only tried when the ATN for complete sentences did not find any interpretation.

More exact control is possible through careful arrangement of the extended rules, so that ill-formed paths through the grammar are searched in order from least to most anomalous. Minton, Hayes, and Fain [38] outline an approach in which each grammar rule has an attached weight, with each parse path then collecting a score. These scores are positive for well-formed interpretations, but negative for ill-formed ones. In this way, they propose that the same mechanism that would allow choosing the most likely among ambiguous well-formed interpretations could also be extended to finding the most likely ill-formed parse. The problem with this sort of approach, of course, is working out such a set of weights and a composition function. The difficulty is that much of the contextual information that must be applied to making the decision about which ill-formed interpretation is most likely will not be available at the point where the parser must make its initial decision, since the actual "cost" of any particular ill-formedness cannot be judged from local context alone.

Weischedel and Sondheimer's meta-rules [52, 55, 56] are an elegant formalism for controlled relaxation of a grammar. Meta-rules are IF-THEN rules whose IF clauses contain diagnostic predicates that test whether the parse state is one to which the rule should apply, and whose THEN clauses make the desired changes in the input string or parser configuration to allow the parse to continue. They are intended as a mechanism for encoding controlled relaxation of the grammar, where rules are only relaxed in the presence of explicit indications that that particular relaxation is appropriate. Careful formulation of the IF parts of the rules can help protect against a flood of spurious parses, as can heuristic choices of where in the sentence to begin applying the relaxations [44]. The rules are intended to capture frequent error types. They are not necessarily syntactic, but since they were first conceived of and organized as an adjunct to a syntax-first grammar, they have a primarily syntactic feel. Again, this is mostly a structure in which a real theory of likely types of ill-formedness could be (but has not yet been) expressed.

There is a tradeoff in all such syntactically-driven systems between power and time. The power in the methods we've looked at so far comes from their attempt to relax or extend their grammars selectively in order to accept those cases of ill-formedness that are most likely, and that thus are fairly easy to search for. However, the more the grammar rules are relaxed, the larger the space that must be searched, which costs time, and the greater the chance of making the wrong correction. One approach that has chosen to allow a large degree of relaxation at the cost of increased search is Trawick's [53] ill-formedness component for the REL system. While he implemented a sequence of increasingly more powerful strategies, in the hope that easy corrections would be found first where possible, his processor before giving up would try replacing in turn each word in the input with all other words in the dictionary that were sufficiently lexically similar, trying to reparse the sentence for each such substitution.

A somewhat different approach is found in the EPISTLE [28] work at IBM, where instead of relaxing the grammar to achieve an exact parse, an approximate parse structure is built by assembling whatever well-formed fragments have been found in a

bottom-up syntactic parse. While highly robust in that it will produce some structure for almost any input, the roughness of the resulting structures represents ambiguity that has only been postponed and will still need to be dealt with by later stages of processing. In addition, in the presence of alias errors, some of the fragments in such a partial parse may not actually belong anywhere in a parse of the intended sentence.

In summary, most syntactically driven ill-formedness methods have focused on controlling the relaxation process so as to find a way past the bottleneck of building an initial representation without creating an impossibly large search space. The use of semantics in such systems for ill-formedness has usually been a direct extension of its use for well-formed input, typically as a filter and analyzer applied to proposed syntactic parses. Most syntactic approaches have so far made little attempt to model or apply pragmatic knowledge for ill-formedness.

3.2 Semantically Driven Methods

To some extent, a parser that is organized around the semantics of key words in the sentence can simply ignore some syntactic ill-formedness issues, since they will not be noticed. Some semantic parsers, indeed, have paid little attention to syntax, and thus would not notice certain types of ill-formedness. For example, the semantic grammar of Burton's SOPHIE CAI system [7], while looking for a predicted case filler, would simply skip over words that did not fit the predicted context, like *very* in the command

Insert a very hard fault.

Even if such a parser does also check syntactic features, because semantic context is guiding the process, these can be easily relaxed in cases where no correct parse can be obtained, letting a partial sense of the meaning be built up from the key words, and ignoring portions that are not understood. The advantage of this approach is that an idea of the semantic content is established early and is thus available for use in the completion of the syntactic parsing. This can help to limit greatly the search for possible corrections, since only possibilities permitted by the semantic context need to be considered. On the other hand, the danger is that this skimming process may miss significant features of the input through premature commitment to particular semantic items. For example, Schank's Integrated Partial Parser [47] for understanding newspaper stories used a measure of contextual interest to determine which words in the input would be parsed, with elements of the text that did not happen to fill slots in the chosen script simply being skipped. He points out that this sort of process can go badly wrong when some unusual feature causes the wrong script to be activated. This sort of "jumping to the wrong conclusion" would require a second, more cautious parsing pass to repair.

Hayes' FlexP system [26] used a bottom-up pattern-matching parser and a limited-domain semantic grammar to handle various sorts of ill-formedness, including interjections and restarted sentences. He relied heavily on semantic patterns embodied in the grammar rules to fill in missing elements or ignore extra ones; for instance, in a message processing domain, the ill-formed input

display new about ADA

would have the noun *messages* filled in when the rest of the input matched the "display message-description" pattern. He stresses that this technique can only be applied in a restricted domain for which such a semantic grammar is possible.

Working with Carbonell, Hayes continued to work on exploiting semantics for ill-formedness with the CASPAR and DYPAR parsers [13, 27]. CASPAR parsed commands robustly by using expectations generated by the main verb. DYPAR was a more powerful version using multiple parsing strategies including recursive case-frame instantiation with equivalence transformations. In their approach, to parse a sentence with an omitted preposition like the following,

Send message John Smith.

the parser uses semantic expectations generated by the verb *send* to determine the proper case marker for the user *John Smith*, ruling out *in* or *of* as incoherent. They go on to make use of a certain amount of pragmatic information by settling on *to* because the destination is a required case for *send* in this pragmatic context. They also use these semantic case-frame expectations to make other sorts of ill-formedness processing like spelling correction more efficient, by limiting the possibilities that the recovery processor must consider to those that are predicted by the semantic expectations. We certainly agree with them that making use of semantic expectations will lead to a more efficient recovery process than one that considers all possible corrections across the whole dictionary. However, we do not agree that a semantic grammar is necessarily the best way to capture those semantic expectations, partly because the model of pragmatic context must be much richer than can be included solely in the case frames of the parser, and also because that style of parser is heavily dependent on the identification of the head verb, so that an error in recognizing that word can lead to a cascade of spurious attempted corrections.

Granger's NOMAD system [19] handled sentences with unknown words in them even when that word was the main verb by using expectations to deduce the word's meaning from context. For the sentence

Enemy scudded *bombs at us.*

the system would first use syntactic and morphological information to deduce that *scudded* is a verb. It would then use a combination of semantic knowledge about the kinds of actions that can link *enemies* and *bombs* along with semantic preferences about which categories of verbs, nouns, and prepositions frequently combine to propose the verb-category *propel*. Granger's semantic preference mechanism [18] in effect encodes a certain amount of pragmatic information, such as that enemies are pragmatically more likely to throw weapons *at* us than *for* us. He also uses a small amount of pragmatics when he tests the proposed interpretations for consistency, eliminating cases of goal violation like an enemy ship peacefully delivering weapons to a friendly ship. However, his pragmatic model is still quite limited compared to the sort of plan-based model we are considering here. It appears to maintain only an average pragmatic context across all possible situations, rather than a model of the current situation. For instance, for the sentence

Contact gained on Kashin.

his system does not have enough sense of pragmatic context to distinguish between (a)

the interpretation in which the speaker gained contact with Kashin and (b) that in which the speaker's current contact advanced on Kashin. Presumably, a richer model of the pragmatic context (whether in terms of scripts or plans) would be able to resolve that ambiguity pragmatically by knowing whether or not the speaker is already in contact with the Kashin, or whether the speaker has a current contact other than the Kashin that could be the subject in interpretation (b). (Of course, naval radio messages may not provide enough context in general to allow the plan tracking necessary for such a model.) Effective use of pragmatic information requires such a richer context model.

Thus while semantically driven systems have made extensive use of the suggestive power of semantics for ill-formedness detection and correction, they have been limited in that they were able to capture and apply very little pragmatic knowledge. Their early and heavy dependence on semantic clues from the ill-formed sentence also implies a danger of jumping to incorrect conclusions, and ignoring or overriding available syntactic or lexical information that supports a different interpretation.

Carberry's work [10] with pragmatically ill-formed input is discussed in Section 5.2. Her approach covered a limited and unusual class of ill-formedness, sentences which are interpretable syntactically and semantically but where that interpretation then fails for some reason in the back-end system, and she used a pragmatic context model based on a tree of domain plans to suggest corrections for such cases of ill-formedness. The approach taken here is closely related to her work and takes it in directions some of which she herself suggested, extending her context model with a metaplan layer and applying it with new heuristics to a different and more problematic class of ill-formedness.

CHAPTER 4

OVERVIEW OF A PRAGMATICS-BASED APPROACH

4.1 Basic Approach: Linking to the Pragmatic Context

Our basic approach can be described as a heuristic search for links between the partial interpretation of an ill-formed input and the predictions of a pragmatic context model. It divides into two parts, first constructing a rich model of the pragmatic context that outlines the space of probable next moves on the agent's part and the queries that those moves might motivate, and second a method for extracting existing syntactic and semantic constraints from the ill-formed query and heuristically combining those constraints with the predictions of the pragmatic model to yield a ranked set of corrections for the ill-formedness. In this section, we describe the approach as a whole, together with presenting an example, to give a general understanding of it. Later sections of this chapter then discuss some of the theoretical implications of this choice of approach and fill in the peripheral elements of the picture like wildcard parsing and the expert advising setting, while the following chapters present in detail the plans and metaplans that make up the pragmatic context model and the heuristics applied in searching for links between the predictions of that model and the partial interpretation of an ill-formed query.

The example we will follow in this section concerns organizing a search for an airliner downed in the North Atlantic. Suppose the agent's first query to the expert is the question

What was its last known location?

followed by the ill-formed query

Where is the case in Iceland?

(That query might be well-formed in a legal context, where *case* would mean lawsuit, but in a naval domain, *case* in the sense of *situation* is the only relevant usage.) Since the system can find no well-formed interpretation of the latter query, it must attempt ill-formedness repair by bringing to bear its model of the pragmatic context.

The system's pragmatic model is based in part on a library of domain plans for how agents can deal with situations of the given type. Since we assume that the system is already aware of the agent's top-level goal, here to arrange search and rescue for the downed airliner, it knows which domain plans are relevant, here including **ship-search, air-search-from-base,** and **air-search-from-carrier**. These domain plans are organized into a classification hierarchy based on their preconditions and effects, so that the plan class **search-for-downed-plane** is divided into the three plan subclasses just listed, and each of them is in turn further divided into subclasses, so that **ship-search** might have subclasses for **navy-vessel-search** and **merchant-vessel-search**. A

node in that classification hierarchy thus stands for a subclass of all the possible plans to achieve the goal, and that node can thus represent a stage in the agent's refining of a proposed solution. Each plan class also specifies those subactions that are required by all plans in that class, so that **air-search-from-base** would include the actions of flying the planes to the area, flying the search pattern, and then getting them back to the base. A deeper plan subclass, **air-search-from-base-with-refueling**, might also include actions for deploying tanker planes and performing in-flight refueling.

The actual pragmatic model includes these domain plan classes as the arguments to metaplans that model the agent's actions in trying to build a plan to achieve the top-level goal. These problem-solving metaplans capture the agent's selection of particular plan subclasses or subactions for exploration as possible elements of the solution, and also the possible queries that the agent might ask as part of exploring those nodes. The particular collection of metaplans used in Pragma models the structure of plan-building and query-generation that typifies this expert advising setting; an equivalent model in other discourse settings would require different sets of metaplans. A particular ***build-plan*** node in the metaplan tree, for example, models the agent's exploration of a particular class of domain plans, and that branch of the metaplan tree then models the agent's possible next moves toward further refining the plan, instantiating its variables, or asking queries to gain information to make such choices. For example, the first query about the last known location of the airliner is a query that the system recognizes as related to the entire class of **search-for-downed-plane** plans.

That model of the space of the agent's possible next moves and queries is the key to applying the pragmatic knowledge to resolving the ill-formedness. The approach is to extract from the ill-formed query a set of partial interpretations that encode as much as can be derived from the syntax and semantics of the remaining sentence by treating each single word of the input in turn as a wildcard. Given the assumption of a single-word error, one of those partial interpretations will be a partial description of the intended query. The technique is to search in the metaplan tree for nodes that predict queries that can be linked to one of these partial interpretations, with each such link suggesting a possible correction to the ill-formed query.

Our proposal for deriving partial interpretations from the ill-formed query based on the single-word error assumption is to replace each word in turn with a wildcard, meaning an empty slot that can take on the syntactic and semantic roles of any known word. Running the parser will then attach to this "wildcard" slot whatever syntactic and/or semantic restrictions are required by the rest of the sentence in order to achieve a meaningful parse, if one is possible. Different parsers might have to implement this approach in different ways, perhaps using multiple passes assuming different syntactic roles, but the final result would be a set of partial interpretations where some of the terms in the logical form would themselves be wildcards, reflecting the unknown semantics of the original wildcard word. For some word positions, naturally, even replacement with a wildcard will not enable any complete parse of the sentence, so those will not produce any candidate partial interpretations. For example, wildcarding *is* in our current example to produce

*Where ***** the case in Iceland?*

does not produce any valid parses. The two word positions that do produce parses here

are *where* and *case*, with the former producing a set of interpretations including those for

What is the case in Iceland?

and

Which is the case in Iceland?

while the latter produces an interpretation where the wildcard substituted for *case* is included in the logical form with the restriction that it stand for something of type **locatable-entity**, so that the query asks about the location of the unique object of that unknown class that is in Iceland. The system would then search for nodes in the metaplan tree that could be linked to those partial interpretations.

The metaplan tree not only models the space of the agent's possible problem-solving moves, it also is annotated with a heuristic component that derives a score for each node in the tree reflective of its relative likelihood as a next move given a particular preceding context. In deriving these scores, the heuristic component draws on information about the shape of the metaplan tree, the particular metaplans involved, and the assumed world knowledge of the agent. These scores are used both to direct the search of the metaplan tree, guiding it first to nodes representing more likely moves from the previous context on the agent's part, and also to rank the solutions found, when there are more than one.

In the current example, the search for nodes that can be linked to the partial interpretations begins at the previous context node which matched the query about the airliner's last known location as relevant to the **search-for-downed-plane** plan class. One of the plan subclass branches within that plan class is the **air-search-from-base** branch, whose plans for flying the planes from the base to the search area refer in their preconditions to the location of the base, so that queries about the location of an airbase are in turn predicted as possible problem-solving moves on the agent's part when exploring that branch. The node containing that query can be linked with the partial interpretation in which the wildcard for *case* is allowed to match any **locatable-entity**, suggesting that the query may be for the location of the *base* in Iceland whose planes might be used in the search.

The other set of partial interpretations of the ill-formed query were for queries about the situation in Iceland, but the system would not find any related queries in the nearby metaplan tree which could easily be linked to them. One could imagine the

WHAT/HOW is the case in Iceland?

form linking with a projected query about the readiness of the squadron based there, which would be one of the relevant queries predicted in the metaplan tree, but only if the system could connect the **situation** entity stemming from *case* with the readiness condition of a squadron. The partial interpretation corresponding to

WHICH is the case in Iceland?

would be even harder to match, since it would require a discourse context that had somehow established a set of situations as accessible discourse entities. The search for

nodes that can be linked to each partial interpretation also scores the resulting matches, and those scores would be used to rank any links resulting from these interpretations against those where *case* was wildcarded, with the likely result that the latter would be preferred.

Thus we see that the basic approach makes use of a rich model of the pragmatic context to map out the space of likely next moves on the agent's part, and then attempts to link the partial interpretations of the ill-formed input that are possible under the single-word error assumption to the queries predicted as likely next moves in the metaplan tree. The strong contextual constraints allow the system to identify the single-word corrections to the input that produce queries most closely related to the pragmatic context.

4.2 Comparing the Linking Approach to Alternative Approaches

As outlined in the preceding section, the basic approach taken here is to derive predicted plan-building moves and queries from the metaplan tree in the neighborhood of the previous context and to search for nodes in that space that can be linked to the partial interpretations of the ill-formed input that are possible under the single-word error assumption. In this section, we compare this approach to the existing syntactic and semantic approaches to ill-formedness.

The major difference between this linking approach and the syntactically driven approaches described in Section 3.1 is in where the initial hypotheses of possible corrections come from. In the syntactic approaches, those hypotheses are generated at the syntactic level, when the parser first builds a structure containing the ill-formedness. For example, a metarule approach [56] to the lack of agreement in the sentence

The ships in Kennedy's task group is in the Med.

would relax the agreement rule between the subject NP and the verb *is* in the parser itself, effectively hypothesizing a correction of *is* to *are*. The semantic and pragmatic processing of these proposed corrected utterances were then treated in the normal way. In contrast, if the linking approach suggested here were applied to this example, it would take the more radical step of wildcarding each word in the input, resulting in at least two partial interpretations, one where a wildcard of *ships* was restricted to some singular entity in the task group, and the other where a wildcard from *is* could be any verb that can take ships as a subject and a PP with *in*. These much more open syntactic possibilities would then be used to search for links in the pragmatic context, with that context being the initial source of suggested corrections.

The argument as to which approach is better depends heavily on which set of constraints offers the most information in the particular class of examples. For cases of ill-formedness that can be treated as agreement errors, the syntactic constraints are very strong, able to propose just one or two possible corrections. In such cases, it makes good sense to start with the syntax, and use semantics and pragmatics to check the results of its proposals. However, there are many classes of ill-formedness for which syntactic constraints offer no guidance. An unknown word, for example, might be assumed to be a garbled version of a known word, but a relaxation metarule could only

propose the replacement of it by every known word of that class, and with alias errors, the problem is even worse, since the parser does not even know which word to try replacing. For such classes of ill-formedness, where there is little that syntactic constraints can offer, it is much more efficient to let the initial hypotheses come from the pragmatic context, and then to apply the syntactic and semantic constraints from the actual utterance.

There is a much closer connection between the linking approach taken here and the semantically driven approaches discussed in Section 3.2. Hayes and Carbonell [13, 27], for example, use the semantics of the main verb in a sentence to provide constraints that are used as early as possible in the correction phase to restrict the set of possible corrections. Their spelling corrector, for example, is restricted to considering possible corrections of a class that their semantic grammar allows in that position in the sentence. Thus, while the generation of possibilities is still done in the parser, they apply the highest level of constraints available in their system very early on as a filter on those possibilities. Granger [19] carried that approach further, using some general pragmatic information so as to create expectations that could suggest corrections even for the main verb in a sentence. However, the original suggestions for fillers were still generated by taking, for example, all the actions that could link a particular subject and object.

Thus these semantic approaches move in the direction of applying higher levels of constraint earlier in the correction process, a move that the linking approach used here takes even further, since here the initial candidates for correction themselves are derived from the pragmatic context. The only replacements that are considered are those that the metaplan model shows are closely connected to the previous context and form likely follow-on queries. Those possibilities in turn are filtered against the syntactic and semantic constraints, rather than syntactic and semantic possibilities being filtered by pragmatics.

This heavy dependence on pragmatics is particularly appropriate to difficult classes of ill-formedness like alias errors where syntax and semantics offer only weak constraints or none at all on the possible corrections. Linking as presented is, in a sense, the heavy artillery required for the hardest classes of ill-formedness. In those cases where syntax or semantics are able by themselves to suggest particular resolutions for the ill-formedness, it may be more efficient simply to test those suggestions against the closely-related pragmatic context rather than to find all contextual possibilities and then apply the syntactic constraints. One possible direction for integrating the two approaches might be to capture such syntactic and semantic preferences in the results of the wildcard parsing, rather than making that a flat set of all possibilities, although that representation would thereby become more complex.

In summary, the rich pragmatic model developed here provides constraints that are important data for any ill-formedness strategy, no matter how that strategy organizes the search of the space of possible corrections. The linking strategy presented here for applying that pragmatic knowledge by using it as the initial suggester of possible corrections that the process then tries to link to partial interpretations of the ill-formed input is a useful and appropriate model for the difficult classes of ill-formedness being considered here. However, it may need to be modified when it is integrated as part of a general-purpose NL system dealing with all classes of

well-formed and ill-formed input so that the strongest available constraints in each case can be brought to bear early in the process to restrict the search space as much as possible.

4.3 Selecting the Level for Linking

Given the outlined approach for these classes of ill-formedness of linking to the pragmatic context model, a further question arises concerning at what level to do the matching that tests for the possibility of a link between a metaplan tree query prediction and the partial interpretation of an ill-formed sentence. The links could theoretically be made between sentences, interpretations, or contexts, but we argue here that the interpretation level is the wisest choice.

The issue is choosing at what level to try matching the ambiguous partial input against the expected goals that are suggested by the pragmatic context. The simplest model of the possibilities comes from representing the problem as a bidirectional search that either works forward from the ill-formed input building toward possible plans that it might be expressing, or else starts from the current plan context and builds toward queries that might make sense in that context. In Figure 4.1,

```
ill-formed  =>  partial ->  plan  ->  previous
sentence    =>  interp  ->  node  ->  context
                  |||
correct     <-  query   <=  plan  <=  previous
sentences   <-  interp  <=  node  <=  context
```

Figure 4.1: Choosing a Level for Matching

the top line, running left to right, represents building forward from the ill-formed input actually found through the set of partial interpretations it might be expressing toward the plan steps of which they could be realizations, while the second line, running right to left, represents the search backward from predicted plan context through the interpretations of possible queries to the set of possible sentences that would be coherent in the context.

The key question of strategy is at what level of representation to try to make the connection between some partial understanding of the input from the top line, on the one hand, and the probable next queries that one can deduce from the plan context model on the bottom, and the main variable affecting the choice is the branching factor of the search in the two directions. While it would be possible to build forward from the ill-formed sentence to any one of the analysis stages, such exploration comes at an increasing cost in ambiguity at each stage, due to the uncertainty created by the ill-formedness. For instance, if a query to a university database about class rank came out

What is Judy Smith's tank?

assuming that *tank* is the error, treating it as a wildcard, and building forward would find partial meaning representations asking about every attribute applicable to students. Each of those possible queries could in turn arise as a query node motivated by various planning nodes in various contexts. Working in the other direction, building backward

from the previous plan context through the probable next plan steps to query interpretations and even to possible sentences also carries an increasing cost in ambiguity at each stage of the process, often a substantial one. There are so many different ways to phrase a question with the same meaning representation, for example, that building backward on the second line all the way to suggested correct sentences in order to do the matching at the sentence level would be quite costly. In the same way, the number of plan nodes that can motivate a given query when considered without respect to any known previous context will also be quite large. Thus, it seems that the combinatorics argue for doing the matching at the level of partial logical interpretations.

Of course, this bidirectional search model oversimplifies the problem somewhat, where the actual goal is to apply the strongest constraints available as early as possible in order to restrict the search space as much as possible. There are cases where information can be derived from one path to help constrain and speed the search in the other. For example, the search in the metaplan tree for plan context nodes that might have generated the current ill-formed query can be made more efficient by making heuristic use of a list of the atomic sentences in the partial interpretations, to encourage exploration of paths whose preconditions overlap heavily with those found in the partial interpretations. Nevertheless, the basic choice of doing the linking at the level of the logical interpretations still seems to hold.

4.4 Wildcarding for Syntactic and Semantic Constraints

In order to do matching against pragmatic patterns, we must find a way using syntax and semantics to derive from the ill-formed sentence some representation for the set of possible meanings it could have, given our assumption that any single word in the input can be relaxed. We want as tight a characterization of this space as possible, one that applies all the syntactic and semantic constraints available. For example, in the sentence

How many courses in moth do I still need?

when we try to relax the word *moth*, syntax tells us that it must be a noun or pronoun. Semantics can further restrict this to nouns of semantic class *X* for which there is some interpretation for the phrase *courses in X*, which would include in this case departments, areas, colleges, and even buildings. (There are also interpretations where the *in* PP is taken as modifying the verb, such as *in total* or *in 86A*, but we ignore these here.) Semantics on its own might be able to prioritize those semantic possibilities, to say that *in buildings* is in general a less likely interpretation than *in departments*, but discriminating among such possibilities will require input from pragmatics. For now, our goal is to find a way to capture all of these syntactic and semantic constraints, building a template that can then be matched against pragmatics.

The search for an effective strategy for computing some representation of the syntactic and semantic possibilities depends heavily on the style of parser in use. The most straightforward way, replacing each word in the sentence with every word in the dictionary, is clearly impractical. Besides, that scheme (or any scheme based on known words) does not offer any help with unknown words, even ones that are synonyms to existing words, and whose semantics are therefore easily representable.

This sort of generate-and-test approach would only be practical if we could find a small set of classes of words, so that we would need to make a test run of the parser only on each class.

A unification style, complex-feature-based parser [48] is one style of parser that we believe would allow a convenient and elegant way of computing a semantic representation for the possible meanings given the single-word error assumption. Such a parser uses unification to determine whether its rules (like (S -> NP VP)) can apply to the input sentence by unifying the terms in the rules with elements of the input sentence. The unification also carries along feature values that enforce syntactic constraints like subject-verb agreement: (S -> NP(number=N) VP(number=N)). In order to determine what sentences are possible if a given word is relaxed and allowed to be replaced with any other word, one can simply replace the given word in the input with a wildcard value, none of whose features are specified. During the course of the parsing, this wildcard will pick up the appropriate restrictions from the various rules that apply. For instance, in the previous example about *How many courses in moth...*, for the NP rule to apply to *moth*, its part of speech would need to be bound either to N or NPR. Each consistent set of rule bindings will yield an interpretation of the sentence in which the wildcard that replaced the suspect word has collected exactly the syntactic constraints that were implied by the rest of the sentence given that interpretation. Because there will in general be more than one consistent set of rule bindings, this would compute a set of partial interpretations for the sentence, with the suspect word's wildcard in each case constrained as necessary. We can use semantics concurrently to filter those syntactic interpretations and to compute a set of partial semantic formulas that represent the possible meanings of the sentence under the single-word error assumption, and those semantic formulas can then be matched against the predictions from the plan tree.

Note that the partial interpretations that result will also contain wildcard items, where the missing constraints due to the wildcard word leave part of the semantic representation unresolved. In the simplest examples, the single wildcard word will produce a single wildcard in the interpretation. For example, one partial interpretation of the ill-formed sentence

The girl named Aan is tall.

in the WML form described in Section 8.3 is shown in Figure 4.2.

```
(assert (tall (iota ?G girl (name-of ?G +W+))))
```

Figure 4.2: Interpretation with Single Wildcard

(Other interpretations would include *the girl named well* and *the girl named yesterday*.) However, there will also be examples where a single wildcard word produces multiple logical wildcards. For example, the partial interpretation of the phrase *the vocation of Fox* shown in Figure 4.3 includes both a wild entity class +W+ and a wild function +X+. (This can be read as the unique entity ?V whose type is +W+ and which stands in the +X+ relationship to the unique vessel whose name is Fox.) Since the semantic encoding of the relationship implied by the word *of* in this case depends on the meaning of the head noun, we get more than one logical wildcard in the partial interpretation.

```
(iota ?V +W+
   (+X+ (iota ?F vessel (name-of ?F Fox))
        ?V)))
```

Figure 4.3: Interpretation with Two Wildcards

In the face of an alias error that makes a sentence unparsable, we propose the use of wildcard parsing to compute a set of possible partial interpretations of the input. In theory, that set could be very large, since each word in the sentence must be wildcarded in turn, and each wildcard parse may find more than one possible interpretation given different substitutions. On the other hand, our explorations seem to show that wildcard parsing is usually able, given the single-word error assumption, to localize the problem fairly well. One reason for this is that many of the words in the sentence, even if wildcarded, will not yield any legal sentence interpretation. For example, if we have the error of *courts* for *courses* in the sentence

Are any Humanities courts offered on weekends?

the only attempts that will yield valid partial sentence interpretations will be those where *courts* and *Humanities* are wildcarded. While *courts* could mean *courses*, *lectures*, or *events*, and *Humanities* could mean *tennis*, *grass*, or *moot*, wildcarding of the other words in the sentence does not yield any valid sentence interpretations. In this way, wildcard parsing itself can help to localize the source of the error.

In addition, there is often enough redundancy present that the successful wildcard attempts can tightly constrain the wildcard slot. For instance, in the sentence

I want so take a course in computer science.

syntax alone can constrain things so that the third word can only be *to*. In a wildcard parse, this result would be discovered naturally when the attempt with *so* wildcarded found only a single rule that would lead to a full sentence parse.

Naturally, all of the constraints encoded in the wildcard parses of an ill-formed utterance could also be discovered, expressed, and applied in other ways, for example, by explicit lists of possible instantiations or by annotations for syntactic and semantic constraints attached to a partial parse tree. Wildcard parsing is proposed here not as a new source of information but as a convenient form for expressing the derived syntactic and semantic constraints, while the unification-based parsing approach appears to be a convenient and elegant way to compute the constraints in that form.

4.5 The Expert Advising Setting

Because the pragmatic model depends in part on capturing in its metaplans the structure of the discourse setting, so that the agent's possible moves can be predicted, a particular setting had to be chosen in which to develop the model. The example discourse setting chosen is the expert advising setting, that of a question answering system able to understand NL queries robustly and to respond cooperatively. This domain has been chosen by many researchers studying cooperative responses and pragmatic modeling (including Carberry [10], who terms it the "information-seeking" domain), both because it is an important target application for NL understanding

systems and because it helps to isolate the problems involved in using pragmatic context for robustness, allowing convenient study.

The expert advising setting is useful for the isolated study of issues related to pragmatic context because it allows certain simplifications compared to the modeling of more general types of discourse. First, the expert is assumed not to be pursuing independent goals, but rather focusing entirely on assisting the user in meeting her goals. Unlike some expert systems, where the user's input is at the level of overall goals which the system then develops a plan to achieve, in an expert advising system, the initiative stays with the user. (This is why the term "agent" has been adopted here in referring to the user of such a system, since her goals are directing the interaction at every planning level.) Second, the expert advising setting reduces the problem of modeling multiple interacting goals on the part of the agent, since the expert consultation usually occurs within a single domain and around a single top-level goal. Third, because the advising consultation happens apart from the agent's action in the domain, the model of the domain context can be taken to be fixed for the course of the interaction. These features of the expert advising setting that make it a useful laboratory for questions of pragmatic context are discussed further in Section 6.1, where Pragma's metaplan model for the agent's behavior in that setting is described.

The expert in such an interaction, of course, does more that just answer the agent's literal queries. As a cooperative participant in the agent's problem-solving process, the expert maintains a model of the agent's goals and of the plan that the agent is refining and instantiating to achieve those goals, and uses that model to determine what helpful information to provide in addition to the direct answers to the agent's queries. In Pragma, just such a model of the agent's goals and of the problem-solving process the agent is pursuing to refine a plan for achieving those goals is used also as the pragmatic context model from which suggested corrections for ill-formed inputs are derived.

The particular problem domains for which examples of expert advising dialogues were considered include a student advising system, cooking advice, and a naval mission planning system, with most of the examples appearing here taken from the latter. In that domain, the system includes a database with information about the positions, courses, and assignments of ships, and their readinesses, both overall (expressed as the letter C plus an integer, from C1 to C5) and for particular missions. The agents in that system are planning naval missions to respond to newly identified needs or to compensate for or correct identified problems in fleet readiness.

CHAPTER 5

DOMAIN PLAN TREES FOR MODELING PRAGMATIC CONTEXT

5.1 Role of the Domain Plan Tree in the Model

If the pragmatic context is to be useful in interpreting ill-formed input, we first need a way to model the pragmatic context of an interaction. The model used in this work includes both domain plans, to capture the intended actions the agent is considering to achieve her goals in the world, and a layer of metaplans which capture the process of building a particular domain plan to achieve those goals from among the possible plans available, including the information-seeking queries that make up the agent's side of the problem-solving dialogue being interpreted. This chapter describes the first of those layers, the domain-level plans. In Section 5.2 we consider the substantial tradition in AI research for using plans to model the agent's intended domain actions, comparing the different models used, and Section 5.3 explains our theory of domain plans. Sections 5.4 and 5.5 then present an approach building on work by Kautz [33] for using classification networks of plans to provide a model that can represent the agent's plans to a flexible degree of detail and that also can model the space of choices for further specifying a plan at each stage of refinement as the agent is constructing that plan.

5.2 Existing Approaches to Plan-Based Models of Pragmatic Context

There is a rich tradition in NL research for using plans to model pragmatic context. Some of the first efforts along these lines came as part of exploring the discourse structure of dialogue. Mann, Moore, and Levin [36] proposed a structure called "dialogue games" to model the pursuit of goals by dialogue participants. A request for help would be seen as an attempt to initiate a "helping-game", with defined roles and expectations for the "helper" and "helpee" participants that served to structure the discourse. Grosz [22] studied apprentice-expert dialogues gathered during assembly of an air compressor and modeled the discourse structure in terms of a hierarchy of focus spaces that was derived from a model of the structure of the task. These focus spaces determined what entities were available as possible referents for noun phrases at each point in the task. Robinson [45, 46] worked in the same domain, constructing a model of the task in terms of a tree of goals and actions, with one branch in focus at any particular time, and used that representation of pragmatic context to interpret vague verb phrases in examples like *I'm doing the pulley now*. Sidner [50, 51] has also worked within this tradition of using planning models for capturing the state of the discourse, which she calls "plan parsing"; in terms of her recent work with Grosz [24], the plan model can be used to define the intentional structure of the dialogue.

Other researchers have used this plan-based style of pragmatic modeling and the extra contextual knowledge it provides in order to generate cooperative responses, meaning answers that provide more helpful information to the agent than just a direct response to a query, and to interpret queries that were for some reason, while not strictly speaking ill-formed, still pragmatically inappropriate or otherwise hard for the system to interpret. The style of plan modeling that has been used in this line of research has been adapted from early AI work in planning. Many of the formalisms build on the STRIPS notation of Fikes and Nilsson [17], in which a plan is made up of preconditions, actions, and effects, where the preconditions are assertions that must be true for the plan to be applied, the actions are either atomic actions or subplans that are required to achieve the plan, and the effects are the assertions that are the result of the plan. A goal can be achieved in this style by successfully executing a plan which has that goal as one of its effects. Before a plan can be executed, its preconditions must either already be true, or must become goals of their own to be achieved. Once the preconditions are satisfied, the actions are executed, by directly doing any atomic actions and by setting up any complex actions as new subgoals to be planned for. Researchers using plans to model pragmatic context have usually adopted some form of this basic AI planning scheme, but with a different focus than that of traditional AI planning. Traditional planning work is interested in constructing a plan to fulfill a certain goal in a given problem situation, so that the issues are things like preventing deleterious plan interaction. Pragmatic modeling, on the other hand, uses the plan database to deduce from the agent's actions the plans that could be motivating them. This bottom-up use of plans leads to different sorts of issues than normal work in top-down planning.

Allen and Perrault [2] used STRIPS-like plans to represent the goals of people in a train station setting. They worked out the necessary heuristics for implementing a simple form of plan recognition, for instance that if a person is doing action A, and A is a step in plan P for goal G, then perhaps the agent is trying to achieve G. They then used their plan model of the agent's goals to provide extra information that was also related to the agent's goals in addition to the direct answer to a question, like supplying the departure time of a train as well, even when asked only for the track number.

Carberry [8, 9, 10, 11] used plan context both to correct pragmatically ill-formed input and to handle intersentential ellipsis. For example, one of her classes of pragmatic ill-formedness covered cases where the agent's query, while syntactically and semantically well-formed, referred to relationships that did not exist in the system's database. Carberry used a model of the agent's plans and goals to suggest a relationship that did exist and that was relevant to the tracked pragmatic context. For example, if only full-time faculty can supervise independent study projects, and a student who is trying to set up such a project asked

What is the classification of Dr. Smith?

the query would be ill-formed because only students have "classifications". However, the system could deduce from its plan model for setting up independent study projects that the "status" of the faculty member is a similar relationship that is relevant, and make the substitution. Carberry also used her plan context model combined with a discourse component to interpret intersentential elliptical fragments like *small bills only* in the example

I want to cash this check. Small bills only.

Carberry used a STRIPS-like scheme for her plan-based pragmatic model, but added some refinements that were important for tracking and predicting the agent's possible plans. For instance, she found it important to add a distinction between "preconditions" and "applicability conditions", where the latter is a precondition that one would not plan to make true. While a student can become a CIS major either by being admitted as a major when she first applies, or by later changing her major, it would not be reasonable for a current student who wants to become a CIS major to plan to be dismissed in order to be able to be admitted as a major, and one does not want the tracking system to propose such anomalous plans. Carberry avoids this by making the state of not being a current student an applicability condition rather than just a precondition of the admit-as-major plan. She also enriched the plan model with other structures like a domain model that can judge the similarity between different entities and relationships, for use in heuristic comparisons between multiple possible suggested plan matches.

Kautz [33], another researcher concerned with plan tracking, has focused on careful, logical definition of the meaning of both plans and hierarchies of plans, and on questions of the adequacy and completeness of plan formalisms. He uses circumscription theory to characterize the logical implications of the assumptions that such plan tracking systems usually make:

- that the system knows about all possible plans,
- that all observed actions are part of some plan, and
- that the desired explanation is the one requiring the smallest number of top-level plans.

The classification scheme presented later in this chapter that classifies plans on the basis of their effects and preconditions and that is used to represent the agent's partially-specified plans was inspired by Kautz's approach.

Pollack [41] has extended plan-based pragmatic modeling by representing in the model what it means for the agent to perform an action as part of a plan for a given goal in terms of the cluster of beliefs that the agent must have for an action to be purposeful in that sense. This allowed her to represent and reason about examples where the agent's beliefs about what actions would achieve a particular goal under what circumstances disagreed with the system's knowledge. She used this richer, belief-sensitive model to propose a mechanism whereby the system could generate helpful responses in such cases. For instance, in the following dialogue

Question: *I want to talk to Kathy.*
Do you know the phone number at the hospital?
Response: *She's already been discharged.*
Her home number is 555-8321.

the system must be able to represent the agent's incorrect belief that Kathy is still in the hospital in order to understand why, for the agent, knowing the phone number at the hospital is relevant to talking to Kathy. Having recognized that incorrect belief, the system can correct it and provide a helpful response, rather than answering the agent's

literal question.

The domain plan model presented here will follow the STRIPS-like approach used by the majority of researchers in plan-based pragmatic modeling. The approach of Kautz in using classification networks of plans in his work in formalizing the process of plan recognition will point the way here to including such a classification approach in our formulation of plans, resulting in a model that allows representing the agent's plans with a flexible level of detail and representing the incremental development of those plans as the agent refines a solution to her problem. Pollack's generalization of plan models in terms of the agent's beliefs is left for future work.

5.3 Theoretical Description of Domain Plans

Before discussing the classification structures built from plans, we briefly consider the definition of plans as action sequences.

5.3.1 Actions and Ground Plans

The form used for domain plans in those systems that build on the STRIPS model depends in large part on the form of the world model, the representation used for the facts and primitive actions of the domain. The domain in such systems is modeled by a collection of logical formulas representing all the relevant facts about the current state of the world. In a blocks world example, such facts include assertions about the robot arm and the status of particular blocks, assertions like those in Figure 5.1.

```
State S1:
   (arm-empty)
   (block B)
   (block C)
   (clear B)
   (on B C)
```

Figure 5.1: Assertions Describe a Blocks World State

A primitive action in such a model represents a change in the state of the world that can be brought about by the planning agent. (Indeed, such systems usually assume that the only possible changes to the state of the world are results of the agent's own actions.) Such changes are modeled by deleting those assertions that are no longer true after the action, and adding any new assertions that the action makes true. For example, one primitive action affecting blocks B and C in the world described above might be as shown in Figure 5.2.

```
Action-Name: pick-up-B-in-state-S1
Delete-List: {(arm-empty), (on B C)}
Add-List: {(holding B), (clear C)}
```

Figure 5.2: Example Primitive Action

A ground plan in such a model is simply a sequence of primitive actions, as shown in Figure 5.3. The delete and add lists for the ground plan as a whole are

```
Plan-Name: reverse-stack-from-state-S1
Actions: (pick-up-B, put-B-on-table,
          pick-up-C, put-C-on-B)
Delete-List: {(clear B), (on B C)}
Add-List: {(clear C), (on C B)}
```

Figure 5.3: Example Ground Plan

derived from those of its actions in the obvious way:

- the delete list of plan P made up of actions A1 and A2 is the union of the delete list of A2 with those formulas from the delete list of A1 that are not on the add list of A2, and
- the add list of P is the union of the add list of A2 with those formulas from the add list of A1 that are not on the delete list of A2.

Using these rules transitively across sequences of actions, one can derive the delete and add lists for any such ground plan.

5.3.2 Plan Schemata

While fully-specified ground plans like these can completely describe the possible courses of action from a given initial state and their effects, they are of little use in a planning system because they need to be worked out afresh for each new task and situation. Ground plans cannot capture the similarity between **pick-up-A** and **pick-up-B**, or the similarity between **pick-up-B-in-state-S1** and **pick-up-B-in-state-S2**. Rather than working with the complete set of possible ground plans for each possible initial state, the individual ground plans are grouped into plan schemata, with variables (written here with prefixed question marks) replacing the constants found in the ground plans. A single plan schema like that shown in Figure 5.4 can thus represent the add and delete lists of many possible ground plans through its many instantiations.

```
Plan-Schema-Header: (pick-up-block-from-table ?block)
Delete-List: {(arm-empty)}
Add-List: {(holding ?block)}
```

Figure 5.4: Example Plan Schema

However, the definition of each ground plan is always in reference to the initial state in which that plan is possible. When we group a set of ground plans into a plan schema, we need some representation for this dependence on the initial state, some test in terms of the schema variables that will tell for a given instantiation whether this plan is or is not applicable in that particular state. To capture this dependence, the plan schema is annotated as in Figure 5.5 with preconditions, formulas in terms of these variables that determine if the action can be performed when instantiated with particular arguments in the current situation. Any instantiation of the variables in a state that makes the preconditions true is warranted to be permissible, and to have the specified delete and add effects.

The dependence of the plan schema on the proper instantiation of the

```
Plan-Schema-Header: (pick-up-block-from-table ?block)
Preconditions: {(arm-empty), (clear ?block)}
Delete-List: {(arm-empty)}
Add-List: {(holding ?block)}
```

Figure 5.5: Plan Schema with Preconditions

variables to produce a ground plan is reflected by including those variables in the schema header, represented as an operator applied to variable arguments, including as arguments the union of all the variables for the included preconditions and actions, as in Figure 5.6.

```
Schema-Header: (pick-up-block-off-block ?top ?bottom)
Preconds: {(block ?top), (block ?bottom),
           (on ?top ?bottom), (clear ?top),
           (arm-empty)}
Delete-List: {(arm-empty), (on ?top ?bottom)}
Add-List: {(holding ?top), (clear ?bottom)}
```

Figure 5.6: Schema with Multiple Arguments

The set of variables included in the header as arguments may be limited to a subset, if the entities omitted are irrelevant to the goals of the plan. Thus the plan header may be a way of specifying a relevant set of entities involved in the goals of the plan. This is certainly true if no effects relative to the omitted entities survive to the delete or add lists of the combined plan schema; for instance, if there are two robot arms available, which arm is used to pick up and then put down a block may not have any effect on the outcome of the combined plan. The header variables are meant to reflect all the relevant dependencies in preconditions and effects.

To apply such a plan schema, one first instantiates the variable arguments to actual terms from the domain; if the preconditions then turn out to be true, the instantiated action is possible, and its execution can be modeled by using the instantiated delete and add lists as before.

A final notational question concerns the types of the variables in the plans. The domain axiomatization often includes among the precondition tests type checks on the variables like (**is-block ?block**). These can be represented either directly, as additional preconditions, or more efficiently by using a typed logic in which the type of each variable is always specified.

5.3.3 Plan Schemata as Subactions in Larger Plans

Just as actions were combined into ground plans, plan schemata, in turn, can be chained together into larger schemata, so that the actions of the larger schema are in turn schemata. When schemata are combined into larger ones, the preconditions of the combined schema are derived from those of the individual schemata: the precondition list of a plan P made up of schemata A1 and A2 is formed as the union of the precondition list of A1 with those formulas on the precondition list of A2 that are not also on the add-list of A1. Such recursive plan schemata are important because one way of phrasing the planning problem is as the search for a plan schema that can be

applied to transform the initial situation into a final situation that meets the desired goals. Since the properties of each plan schema can be derived from those of its constituent actions, building a complete plan from these recursive schemata is formally equivalent to building it directly from the primitive actions, but it can be much more efficient, since productive sequences of actions can be stored together instead of being recomputed each time.

Given a library of plan schemata, planning for a goal could be implemented by searching the known set of plan schemata for those whose effects (add and delete lists) unify with the desired goal or some clause within it, that is, where a substitution of constants exists for the variables in the plan schema under which the effects list of the schema achieves the goal. If instantiated schemata can be found whose preconditions are currently satisfied, those are admissible plans for the given goal. Failing that, instantiated schemata whose effects will achieve the goal may be found with preconditions that are not currently satisfied, but that can in turn become goals to be planned for.

Note that this model of plan schemata so far is simpler than that used by some other researchers, who have developed more complex structures by building in heuristic distinctions to make planning or plan-recognition easier. Distinctions often made that we have ignored here include separating the effects list into a goals list, meaning effects for which it is reasonable to invoke the plan, and other effects, which may be thought of as side-effects that the plan does make true but that are not reasonable goals for invoking the plan. Such distinctions are more important to systems that are doing plan recognition, where the goal is to guess from an action sequence which plans may have generated it. In Pragma, the space of possible plans is instead expanded downward from a known top-level goal, so that the subactions of any particular possible plan are already expanded as the tree is built.

5.4 Classifying Schemata for Modeling Plan-Building

The ground plans and plan schemata described in the previous section are such as might be used in a planning or plan recognition system, but the purpose for using plans in this work is a bit different. In this work, the goal is to use these STRIPS-based domain plan schemata to model the process of considering alternatives and constructing a plan to be used in problem solving. Thus, we need to consider what kind of structure of plans will best represent the space of choices that the agent is considering and the decisions she is making in such cases.

In traditional planning, the primitive actions and situation model are taken as given, and the important question is to work out how those actions may be combined into plans that achieve given goals. The issues there focus around the efficiency of the resulting plan, possible interactions between plans, and intelligent search of the plan space to avoid undesirable interactions. Our goal, however, is to model the plan elaboration process of an agent who knows the possible plans, but does not know the situation. Thus, we are not concerned with building novel plans or working out interactions between plans, but with choosing between known plans on the basis of unknown facts.

In this section, we describe an approach that classifies plan schemata based on their effects and preconditions, and uses nodes in the resulting classification structure to represent the agent's partially-specified plans during this plan-building process.

5.4.1 Plan Classes

To model this process of elaborating plans, we define plan classes, which specify sets of plans in terms of the preconditions and effects that they share. It is these plan classes that will be used in our model where other systems would typically use plan instances. For example, in our blocks world domain, one could define a plan class for those plans that build a three-block tower from blocks A, B, and C, as in Figure 5.7.

```
Plan-Class-Header: (build-3-tower A B C)
Preconditions: {(block A), (block B), (block C)}
Add-List: {(on A B), (on B C)}
```

Figure 5.7: Example Plan Class

This plan class description defines the set of plans whose effects include adding those two add-list propositions. If we consider a situation where B and C are clear, but block D is on A, then this set would include at least one plan in which D is first removed and then B and A are stacked on C and another where B is stacked on C first, then D removed and A stacked. In more complex environments, the plans that satisfy a particular plan class description may differ not just in order of actions, but in which actions are included and even which entities are affected by those actions, as long as they meet the shared plan class specification.

A couple of formal concerns must be disposed of concerning the minimality of the set of plans defined by a plan class description. Because a plan class like **build-3-tower** in Figure 5.7 is understood to include all plans that meet the given precondition and effect list restrictions, it also might be thought to include in any situation a large number of plans that contain more preconditions than necessary to produce their effects, extra preconditions unrelated to the actual actions in the plan. But such "plans" would not fit our definition, where the preconditions must be deduced from the actual preconditions of the constituent actions. Another possible problem could be that a plan class description might be overly restrictive, including in its preconditions a fact that is a precondition to only some of the plans that actually could achieve the given effects, and thus not including some perhaps simpler means of achieving the given goals. However, because plan classes in this system are defined by classification over preconditions, beginning with a plan class that adds no precondition restrictions of its own to those deducible from the set of all plan schemata that could achieve the given goals, any such overly restrictive plan class would have as a superclass a plan class that includes only the necessary preconditions for the necessary actions to achieve the goal effects.

5.4.2 Plan Class Trees

This section presents examples of the use of a tree of plan classes to represent increasingly more specific planning solutions to achieve a given goal. The preconditions of a plan class describe those facts that must be true about the situation for the plans in that plan class to be applicable. The preconditions of a plan subclass, then, are used to specify a relevant subset of the possible situations from the parent plan class and the set of plans feasible in that restricted set of situations. Thus restricting a plan class by adding an additional precondition produces a subset of the original set, and this process is what is used to model the plan elaboration process.

For example, imagine a belfry where the bolts holding the bells are rusted. One bell has actually fallen, breaking through the center of the floor under the bells to the floor below. Taken as a blocks world domain, the task is to use blocks and possibly beams to support the remaining bell. This can be achieved either by building up blocks straight from the floor below or by bridging the hole with a beam and building from there. The initial plan class that represents all plans achieving the desired effect is shown in Figure 5.8.

```
Plan-Class-Name: (support-bell ?bell)
Preconditions: {(bell ?bell)}
Add-List: {(supported ?bell)}
```

Figure 5.8: Plan Class for Support-Bell

(Assume that the **supported** predicate is in turn defined in terms of being **on** something that is eventually **on** the ground.)

We can define a plan class as in Figure 5.9 that is similar to **support-bell**, but that adds the precondition (**beam ?beam**).

```
Plan-Class-Name: (support-bell-using-beam ?bell)
Free-Vars: {?beam}
Preconditions: {(bell ?bell),
                (beam ?beam),
                (available ?beam)}
Add-List: {(spans-hole ?beam), (supported ?bell)}
```

Figure 5.9: Plan Class for Support-Bell-Using-Beam

It is clear that every plan included in **support-bell-using-beam** is also included in **support-bell**, so that this new, more restricted plan class may be considered a plan subclass of its parent. Because the effects and preconditions of a parent must also apply to all the plan subclasses in its subtree, we can in fact describe this plan subclass more succinctly as in Figure 5.10 by specifying only the additional restrictions, and then including a pointer to its parent.

We can build a tree of plan classes in this way, defining a hierarchical structure on the set of all plans that have the desired effects on the basis of the situations to which they apply. Such a tree of plan classes can then be used to model the process of plan elaboration by an agent who knows the plans initially but does not know the facts of the current situation. The planning process begins at the unique plan

```
Plan-Class-Name: (support-bell-using-beam ?bell)
Parent: (support-bell ?bell)
Free-Vars: {?beam}
Preconditions: {(beam ?beam), (available ?beam)}
Add-List: {(spans-hole ?beam)}
```

Figure 5.10: Plan Class Omitting Inherited Preconditions

class that specifies the desired effects and a null set of preconditions. As the agent searches within that class for a concrete plan whose preconditions are satisfied in the current situation, it is the plan subclass structure of that class that the agent ends up exploring.

While the total number of plan classes in reasonable domains is huge, being a powerset of the powerset of actions, an actual system can only consider a limited library of them. That collection of plan classes is a heuristic selection limited first to the likely goals that arise in the domain, since there is no point in storing data about plan classes for goals that agents do not actually pursue, and limited also to preconditions that are relevant decision points in elaborating plans for those goals. For example, in the ships domain, there are a number of plan classes defined whose common effect is the change in location of a vessel from one place to another. These plan classes could all be defined as subclasses of the plan class **sail**, shown in Figure 5.11.

```
Plan-Class-Name: (sail ?ship ?from ?to)
Preconditions: {(vessel ?ship),
                (position ?from),
                (position ?to)}
Delete-List: {(location ?ship ?from)}
Add-List: {(location ?ship ?to)}
```

Figure 5.11: Plan Class for Sail

Subclasses within that class might be based on preconditions like the propulsion-type of the vessel or the amount of fuel on board, as in Figure 5.12.

```
Plan-Class-Name: (sail-nuclear ?ship ?from ?to)
Parent-Class: (sail ?ship ?from ?to)
Preconditions: {(propulsion-type ?ship ?nuclear)}
```

Figure 5.12: Plan Subclass of Sail

As with plan schemata, the plan classes in the library are given plan class headers, consisting of a unique name for the plan class combined with a list of unbound, existentially quantified variable arguments, in which case the plans that belong to the class are those that match its restrictions according to some substitution. For example, **(build-3-tower ?X ?Y ?Z)** is the header of a plan class schema that models the set of plans for stacking up whatever three blocks are supplied as its arguments. One way to restrict such a plan class is by instantiating one of its arguments to a particular value, so that **(build-3-tower A ?Y ?Z)** would describe that subset of the former plan class in which block A ended up on top of the tower. (This is equivalent to adding the precondition (= ?X A).)

These plan class headers resemble an operator applied to arguments, and indeed, while a Pragma metaplan tree is being built, this list of argument variables does serve as a means for passing down from plan class to plan subclasses and subactions the values that need to be instantiated for the desired effects to apply. Note that if the variable list in the header does not include some of the entities affected by the plan, their omission amounts to the adoption of a perspective on the plan viewing it in relation to a goal that is only a subset of its total effects. The choice of which variables to include in the header thus depends on the goals for which the plan class is usually considered. Unbound variables that do not appear in the plan class header are called "free" variables (and are declared as such), implying the heuristic judgment that the plan classes that make use of this plan class as a subplan or subaction will not care what particular value is used to instantiate those variables. For example, the car in a **drive-car-to-repair-shop** plan would usually be relevant at the higher level, so that a plan class header for that plan would include a variable for the car, while the header for **take-taxi-to-airport** typically would not bother specifying which car, even though the effects of both plans do change the values of the car's location. Instead, the taxi would be introduced as a free variable in the lower plan, thus becoming available for use in lower-level plan classes like **enter-auto**.

We can also constrain a plan class definition by adding an "actions" field to the plan class specification that can take other plan class headers as values. The effect is to embody in the higher-level plan class definition the effect and precondition restrictions associated with the action header's plan class. For example, **sail-conventional** might have a subclass **sail-with-refueling** as shown in Figure 5.13 that includes a **refuel** action as well as **sail** actions for the portions of the voyage before and after the refueling, and which would also need to introduce a variable for the oiler used in the refueling.

```
Plan-Class-Name: (sail-with-refueling ?ship ?from ?to)
Parent-Class: (sail-conventional ?ship ?from ?to)
Free-Vars: {?oiler ?rendez-vous}
Preconditions: {(oiler ?oiler),
                (position ?rendez-vous)}
Actions: {(sail ?ship ?from ?rendez-vous),
          (refuel ?ship ?oiler ?rendez-vous),
          (sail ?ship ?rendez-vous ?to)}
```

Figure 5.13: Subclass Introducing Subactions and New Variable

(Of course, a fuller model would include the current location of the oiler and its trip to the rendez-vous.) Including actions in the plan class definition thus serves to restrict the scope of the plan class in particular ways.

Note that the Pragma implementation of this plan class tree model substitutes a type system with automatic inheritance for the explicit type assertions used in these examples. In addition to computational efficiency, the type system is used as a rough heuristic to mark classes of domain knowledge that the planning agent can be assumed to know. While our picture of the planning situation assumes in general that the agent does not know the values of the assertions defining the current state of the world, there are still some levels of knowledge that we can assume. For example, in our ships domain, we can assume that the agent would recognize that "Enterprise" was a ship,

and "30N-127W" a position, even though we would not expect her to know whether or not the Enterprise was currently at that position. The plan class tree ideally should include all the plans that the agent may believe to be possible given her world knowledge, whether or not they are actually possible. Therefore, the only restriction on plans in the initial plan class is that they be instantiations of some plan schema that unifies with the desired goals, so that the set of possibilities depends only on the goals and the plan definitions, which the agent does know. However, we do not want to clutter the plan class tree with plans the agent knows are impossible. If type knowledge is treated just like other preconditions that the agent may not know, then the space of apparently plausible plans that must be considered grows tremendously, including many nonsensical plans. Thus it is a useful first cut to treat the information coded in the type system as assumed knowledge, while not presuming agent knowledge of any other preconditions.

If this heuristic use of the type system to limit the space of possible plans that must be considered is to work correctly, the type system must be defined to an appropriate level. There is always a question of how detailed to make the types, whether at the level, for example, of vessel, destroyer, or "Means class destroyer located at position P with fuel level F". This heuristic will only be useful if that level is set at a conservative estimate of the knowledge that can be assumed on the agent's part, since the planner will not be able to represent the agent's consideration of reasonable plans outside that space if the type system includes too much of the semantics of the domain, and will represent many impossible solutions if it includes too little. In Section 7.2.2, we will present an approach to a more detailed model of the agent's world knowledge, and show how to make use of that model to help control plan tree growth and thus the space of possible plans to be considered.

5.5 Plan Class Trees as a Plan-Building Model

In this section, we discuss why plan class trees are well-suited to the goal of modeling the agent's plan-building process, comparing them to more traditional models of plan context and showing how the differences are motivated by the goals of the modeling in this case. The most important feature in a model of plan-building is that partially-specified plans be easily represented to varying degrees of detail. This is needed partly because the agent's actual state while working out a plan is to have a partial specification of a plan for the task, with certain choices about strategy made and others still open. The representation must be able to handle many such partially-specified plans, as the agent considers different approaches. The second reason why a partial specification mechanism is important is because the expert often has only partial knowledge of the agent's plan-building process. The expert knows only what the agent reveals through her queries or what the agent decides to directly tell her, to help her build a good plan model, and that information is usually not enough for the expert to deduce the agent's plan fully even to the extent that it is clear in the agent's own mind. There again is reason why the representation needs to be able to handle partially-specified plans, to represent the expert's partial knowledge of the agent's intentions. The key advantage of plan classes and plan class trees is exactly that they do allow for that kind of partial specification.

Not only do plan classes allow for partial specification of plans, the

organization of the plan class hierarchy is also structured so that the subclass relationships group together the sets of plans that are needed for representing plan-building behavior. In planning in the STRIPS tradition, the purpose is to discover combinations of actions that will achieve the given goal. One assumes that the actions and world state are known, but the plans unknown, and tries to derive novel plans, so that the plan-action structure is key. Any point within that search space is a complete plan specification, though it may or may not be one that achieves the goals. In plan recognition, the context is the opposite one of deducing, given a sequence of actions, which plans the actor might be engaged in. It was for this purpose that Kautz [33] worked out a plan-classification approach orthogonal to the traditional planning tree of plans and included actions. This classification grouped together sets of plans that contained the same action subsequences, and thus that belonged together as possible explanations for those actions.

Modeling plan-building is yet a third use for plan networks. Like recognition, it shares the need for representing ambiguity as sets of possible plans, but in plan-building, that ambiguity is across sets of plans that achieve similar goals in similar circumstances, rather than across sets of plans that happen to make use of the same actions. In plan recognition, **express-thanks** and **complain** belong together as plans that could include the action **write-letter**. In plan-building, the interesting set is all ways of achieving the particular **complain**, perhaps including **write-letter** and **make-phone-call**. Plan class hierarchies thus represent the structure most needed for modeling plan-building.

Like the plan recognition lattice, the full structure of plan classes is too large to be computed in advance, but while plan recognition algorithms can work incrementally, computing, given each new action, the appropriate plan subsets explaining it along with the previous data, a plan-building model, because it works downward from a defined overall goal, seems limited to exploring the space defined by its library of plan classes, since deriving new plan classes would require computing the effects of action sequences and picking criteria by which to subset the results. Still, although the model seems limited to the stored classes, it has unique advantages in terms of expressing the kinds of partial plan structures that can describe the plan-building process effectively.

In the actual pragmatic context model used in Pragma, domain plan classes, meaning nodes in the hierarchy presented in this chapter that partially specify the agent's current domain plan, serve as the plan arguments for the metaplans that are used to model the agent's plan-building moves. Thus the ***build-subplan*** metaplan will be seen to represent the selection by the agent of one of the possible subclasses of the previously current plan class. The partial specification permitted by the plan class representation allows a single plan class when used as a metaplan argument to represent a full plan context.

CHAPTER 6

METAPLANS TO MODEL THE PROBLEM-SOLVING CONTEXT

6.1 The Problem-Solving Context in the Expert Advising Setting

In the previous chapter, we discussed the use of plans to model the agent's domain-level intentions. But while the domain plan level captures a good deal of useful information about the pragmatic context, there is a level of structure to the agent's purposeful activity in the expert advising setting that it does not capture, which we refer to here as the problem-solving level. In this chapter, we go on to present a mechanism for modeling the agent's goals and plans on the problem-solving level. It is this level that will allow us to model the connections between the underlying domain plan being developed and the agent's observable behavior, and thus to be able to use predictions of likely next steps to suggest corrections for ill-formed inputs.

Consider, then, an agent involved in a complex task that requires selection of a plan strategy, a possibly-novel combination of known subplans. Because the task is too complex for any single stored plan, the agent does not have a canned plan already worked out for dealing with the situation. Instead, she has to design the plan as she goes, and this planning task is logically separate from the domain level tasks that are required in order to achieve the goal. That planning task includes activities like:

- constructing composite plans from known subplans for goals where no complete plan is known in advance,
- testing the feasibility of possible partial plans by seeing if their preconditions are satisfied in the current situation, or if other plans exist that could cause them to be satisfied,
- performing actions whose goal is to learn facts that will contribute to the plan building process, such as working out the results of particular compound plans or asking about the value of unknown environmental features that affect the preconditions, and
- comparing the effects of alternative feasible plans to choose the most desirable.

It is easy to see that that kind of plan-building activity is not captured by domain-level plans, since it often requires representing multiple alternative domain plan structures and modeling operations on them.

As explained below in Section 6.2.3, various researchers have proposed models that do allow representing this plan-building level in one way or another. Wilensky [59], in particular, developed a detailed model of plan building phrased as metaplans, turning the now traditional mechanism of plans, goals, and preconditions to

this new task of representing the agent's actions on this plan-building level, rather than on the domain level. This new level of planning is termed "metaplanning" because these plans have arguments whose values are complete domain plan structures. For example, Wilensky's metaplans included plans for comparing or combining different domain plans or for simulating (deducing in an abstract model) the effects of a proposed domain plan to determine its feasibility.

In this section, we propose a metaplan model that builds in many ways on Wilensky's and other earlier work, but that is designed particularly for representing the pragmatic context in the expert advising setting. The model will be specialized in that it will try to focus on the elements of problem-solving context that can be deduced in such situations from the agent's observed behavior, and ignore elements that are not normally evident and thus would usually have to be left ambiguous anyway. We will show how this problem-solving metaplan model allows us not only to model an agent's given state in an expert advising context, but also to predict from a given state the space of succeeding problem-solving actions that are consistent with the model, and even to judge heuristically which succeeding actions are most likely in the sense of being most closely related to the previous context.

The expert advising setting is in some ways different from most contexts that involve problem-solving behavior, since the agent's problem-solving activity is isolated in this setting, while it usually occurs interleaved with the actions represented by the domain plans, as when an agent realizes that while there are many different orders in which to arrange the errands on a shopping trip, they all begin with finding the car keys and going to and starting the car, and thus begins to perform those domain actions in parallel with the further plan-building activity required to determine a satisfactory order for the remainder of the steps in the plan. Indeed, this interleaving of metaplanning and domain planning is itself a metaplan strategy for increased efficiency, since the activities can then often be carried out in parallel. Thus, a complete model for purposeful action would need to be able to represent such interleaving of metaplans and domain plans, the effects that motivate it and the strategies used to achieve it.

However, the discourse setting of expert advising has the effect of eliminating this interleaving. During consultations of the sort that we are modeling, domain level action has been suspended, so that the domain planning state can be taken to be fixed. Thus the agent's current actions during the consultation are concerned only with the plan-building activities of gathering information and creating the plan that will later be executed. This has the dual effect of simplifying the modeling for domain level plans, since conditions can be assumed not to change and since there is thus no need to represent the state of execution of the domain plans or to update one's world model due to changes brought about by them, and of increasing the importance of the problem-solving metalevel, since that is the level at which the agent's attention is currently solely focused, as she builds possible plan structures, tests them for feasibility, and compares their effects. Thus the expert advising setting not only requires modeling of the agent's problem-solving behavior on the metaplan level, but also naturally draws attention to that level as the primary focus for the agent and thus for the system at this point.

That process of plan building for problem solving on the agent's part is the

subject of this section. It is an important part of this thesis to show the advantages of modeling that process explicitly, just as the agent's domain plans and goals are modeled explicitly. This explicit model of the problem-solving level allows the system to track the agent's metalevel approach to plan building, and thus enables more accurate predictions about the next steps the agent is likely to take in building the plan than a model that represents only the domain plan level.

6.2 Existing Models of the Problem-Solving Context

In this section, we examine existing approaches for dealing with the problem-solving context in expert advising discourse, some approaches that work directly from the domain plans, some that use other structures parallel to the domain plan tree for tracking the discourse context, and finally some that have used metaplans in a way similar to that proposed here to capture particular features of discourse structure.

6.2.1 Working Directly from the Domain Plan Tree

The original users of plan tree models for cooperative responses relied primarily on the domain plan tree as their context model. Information about the patterns of the agent's movements in that tree and about the connection between nodes of that tree and the agent's queries were directly implemented in the system's code for exploring the tree and selecting what nodes would match which queries. No separate structures were maintained to model metalevel context or its implications for the derivation of queries from the domain plan nodes.

Thus, in Allen and Perrault's [2] system, plan inference rules were used to derive from a given query the domain plan that was its likely motivation, in that case, meeting or boarding a train. The system then extracted from its knowledge of that domain plan the obstacles like not knowing the gate at which the train was scheduled to arrive that might be preventing the agent from executing the plan, and generated helpful responses by supplying that data to the agent. The processing in their approach, then, went from the query to inferring a branch of the plan tree to deriving obstacles and formulating the response. Since there was no prediction from one query to the next, they did not model possible next moves in the plan tree or predict from such moves likely follow-on queries. The only movement in the tree considered was the search of the subplans of the selected branch for the identification of obstacles. Since their agents were assumed to be pursuing a single plan, there was no need to represent consideration of multiple alternative plans for comparison or choosing between them. Thus the domain plan tree itself provided an adequate context model for their purposes.

Carberry [10] later expanded the range of this domain-plan-based approach in various ways. Building on the work of Grosz [22], Robinson [45, 46], Sidner [50], and others who had studied more extended plan-based models to represent the task structure of discourse in order to help identify discourse focus, resolve anaphora, and distinguish intended from unintended responses, Carberry developed a mechanism that, like Allen's, deduced a plan context from a single query, but, unlike his, then

maintained that context and tracked the agent's progress through the plan during the ensuing dialogue. Her system included a substantial body of heuristic rules that predicted the most likely movements in the plan tree, rules that guided the search for the related node assumed to be motivating a follow-on query. That ability to carry the context model between queries allowed her to make use of it to handle cases of pragmatic ill-formedness and intersentential ellipsis, both examples where the interpretation of the current query depends on information available only from context. Her basic representation for pragmatic context remained nodes in a domain plan tree, although she also enriched her context model to handle discourse features, as we will see in the next section.

6.2.2 Discourse Models for Problem-Solving Dialogue

Researchers in discourse have suggested various structures for representing pragmatic context which could also be applied to problem-solving dialogue in the expert advising setting. Grosz and Sidner [24] have outlined in their theory an important classification of the kinds of discourse structure that need to be modeled. In their model, the three kinds of structure are the linguistic structure, the intentional structure, and the attentional state, where the linguistic structure is the obvious one of sentences, interchanges, and paragraphs, affected by clue words like *first* and *finally*, the intentional structure refers to the public purposes motivating the speakers, and the attentional state describes the current focus of the discourse and mediates reference issues. They model the attentional state as a stack of focus spaces which can be pushed or popped based on changes in the linguistic and intentional structures, while the intentional structure is modeled by the purposes assigned to each discourse segment along with other purposes which may dominate it or depend on it. In their terms, the domain plan trees that others have used to model pragmatic context are particularly well-organized examples of intentional structure, which is why they arise in task dialogues or problem-solving contexts, where the discourse is closely tied to the tree-like intentional structure of a single task. It is only in that restricted class of domain that the plan tree itself can serve as an adequate model for the intentional structure.

We see in Carberry's [10] work also signs of this stretching of the domain plan tree model in order to be able to capture and predict other sorts of utterance than direct queries about plan preconditions. She maintained in addition to the plan tree a stack of discourse purposes, and that stack combined with the plan context to predict possible utterances. For example, the stack kept track when the previous move had been a question asked by the system, and could predict the agent's possible responses like answering the question or seeking clarification. The context maintained by this discourse stack was used to help analyze the various roles that elliptical utterances can play based on their discourse context. Thus she was able to predict elliptical responses like **express-surprise-at-question** when the agent responds indirectly to a question from the system:

System: *Do you want to take CIS 200 at 8:00 M-W?*
Agent: *At 8:00?*

Thus while Carberry's model of intentional structure was still a domain plan tree, her additions to the model allowed her to capture other features of the discourse context and thus to predict more complex kinds of interaction than the plan tree itself could

support.

The pragmatic model presented here also goes beyond the plain domain plan tree approach, but rather than expanding the discourse coverage, the focus here is on higher levels within the intentional structure. We assume here that the expert advising setting is one where the intentional structure of the discourse is closely enough tied to the plans involved in achieving a single goal that a task structure based on that goal can be an adequate model of the intentional structure. We also limit ourselves in this work to that subset of discourse in this setting involving direct queries from the agent for information, so that the effects of the attentional state and discourse structure will follow the intentional structure closely enough that no separate model will be needed. The new focus in this work, then, comes within the model of the intentional structure, in expanding the model based on domain plans to one that can capture and predict the different problem-solving structures that agents may employ in refining and instantiating a plan for a given goal. A full NL understanding system for expert advising would naturally require both the richer discourse model pointed to by Carberry and by Grosz and Sidner and the model of the problem-solving intentional structure of discourse in that setting presented here.

6.2.3 Uses of Metaplans in Pragmatic Modeling

There have also been earlier efforts to capture either problem-solving structure or discourse structure by using metaplans. In this section, we compare these uses of metaplans with the version proposed here.

Wilensky [59] has presented a very extensive model for a wide range of purposeful activity in terms of many layers of interacting metaplans, built on top of the layer of domain plans. He assumes that the agent has access to a library of canned plans for many situations, and that many goals are achievable by using those prestored plans, a strategy encoded in his **use-normal-plan** metaplan. Novel situations may arise, however, requiring other metaplan approaches, like **change-circumstance**, which tries to modify the situation so that a standard plan will apply, or **simulate-and-select**, which simulates the effects of possible sequences of actions in order to work out an appropriate plan. Adjusting to handle plan interactions is also modeled here by metaplans, so that the **merge-plans** metaplan tries to combine groups of domain plans that can be performed more efficiently together, while **try-alternative-plan** describes one possible response when two plans end up colliding, trying to replace one of the plans with a different plan for the same goal that does not collide. His model is, altogether, an impressive formulation of the logic of plan building in terms of metaplans.

The metaplan model presented here is in many ways an application and development of Wilensky-style metaplanning focused on the incremental refinement and instantiation sort of plan building typical of agents in the expert advising setting. The two main differences between Wilensky's model and the metaplans used here are in the sort of planning captured and the number of layers of metaplanning allowed. First, then, while Wilensky does model many of the activities that are included in our problem-solving domain, the primary focus in his work is on the effects of plans and interactions between plans as they are assembled into larger structures. In the approach

taken here, by contrast, because the arguments to the metaplans here are plan classes rather than individual ground plans, the primary focus is on the substructure of the plans, and the gradual refinement of the plan class into a particular plan. The elaboration of the query metaplans in this model also represents a different kind of application of them than found in Wilensky. Second, the comparative simplicity of the metaplan structure in the expert advising setting, since the agent is known to be pursuing expansion and implementation of a single goal via non-interacting plans, has allowed the current model to be implemented using a single level of metaplanning, so that the actual context tree nodes contain metaplans whose arguments are directly domain plans, rather than allowing the deeper recursion of Wilensky's model, where metaplanning could continue to an arbitrary number of levels.

Other researchers have used less ambitious versions of metaplan models in a way more focused on modeling discourse structure. Mann, Moore, and Levin's early work on *dialogue games* [36] can even be seen as one type of metaplan, a plan-like structure for the discourse which included roles filled by the domain plans.

A more recent use of metaplans to model discourse structure is Litman's work [35]. She used metaplans like **introduce-plan, continue-plan,** and **modify-plan** to organize the plan tracking process during a dialogue. For instance, in her analysis of the editing dialogue in Figure 6.1,

User: *Make a subc [subconcept] of Pastime called Sport.*
System: *OK.*
User: *Make an iconcept [individualized concept] of Sport called Golf.*
System: *OK.*
User: *Is there a concept called mailing-address or something like that?*
System: *There is no concept "mailing-address".*
What do you mean by "or something like that?"

Figure 6.1: Example Dialogue from Litman

the question about a *mailing-address* concept is an example of **introduce-plan** that begins a clarification subdialogue, interrupting the continuing editing plan context. That subdialogue is in turn interrupted when the system's **introduce-plan** requests a clarification of the clarification. Each of these metaplan-modeled discourse operations like **introduce-plan** can create a new context centered around a different domain plan. Litman thus uses a small collection of metaplans to manage a stack-like discourse context representation that does an effective job of capturing the stack-like structure of such clarification dialogues.

The main differences between Litman's work and the approach taken here are again due to which features of the pragmatic organization are being modeled in the metaplans. For clarification subdialogue structure, she uses a small set of fully-recursive metaplans that may introduce domain plan contexts unrelated to the previously prevailing one, while to represent the problem-solving structure in the expert advising setting, we here use a larger collection of non-recursive metaplans which can each change the focus in the domain plan tree by at most a single link.

Another contrast between Litman's style of metaplan model and the one presented here is that her metaplans are used to manage multiple unrelated domain

contexts, while in our use of metaplanning, which depends on one tree for the entire context, separate contexts can only be handled if they can be recognized as sibling portions of metaplan tree structure with a common metaplanning ancestor. In general, changes of context that imply little relationship are not modeled well by our version of metaplans, like the new context introduced into a phone conversation when the child of one participant starts tossing a baseball around the living room, which can become a new context for queries in the conversation but is not related in any meaningful way to the previous context. The single-tree method is useful for modeling in greater detail and with more predictive power that portion of an interaction that is directed toward a single domain goal, which then serves as the root of the metaplanning tree, but it does not capture the stack-like discipline that can govern interruptions.

6.3 Overview of a Metaplan Model

Building on this earlier work using metaplans, we consider here how to model the problem-solving level of an expert advising interaction in those terms. This section describes the structure of the metaplan tree including the relationship between the metaplans and their domain plan arguments, and introduces the four classes into which the metaplans are grouped. The following sections will then describe the metaplans of each group in detail.

6.3.1 The Metaplan Context Tree

In order to model the possible problem-solving behaviors of agents pursuing goals in an expert advising setting, we propose embedding the domain plan classes that represent the decomposition structure of possible plans as arguments in a tree of metaplans that capture both the plan refining and variable instantiating moves open to the agent and also the different queries that might be motivated by consideration of those moves. These problem-solving [PS] metaplans are formulated in a style similar to the domain plans, except that their actions refer to metaplanning operations that the agent employs in constructing a plan to achieve the domain goal, operations like choosing a subplan or instantiating a plan variable, and their precondition predicates are facts about the domain plans like (**sub-action-of replace-ship-plan sail-action**) rather than facts from the database representing the state of events in the world.

The same theoretical classification structure described for the domain plans in Chapter 5 also applies to the metaplans, although because the role of the metaplans in the model is different, less use is made of that structure with metaplans than with domain plans. With domain plans, the plan class in a context represents the agent's partially-specified plan at its current level of refinement, while at the metaplan level, a current context in the discourse is always represented by the fully-specified metaplan, a leaf of the metaplan class tree, that matches or links with the current query. Thus while the class structure of the metaplans is used to define the internal structure of the tree and thus the space of possible metaplanning moves that is searched, any particular move by the agent maps to a fully-specified metaplan, though the arguments to that metaplan may involve a domain plan class that is only partially specified.

By convention, the PS plan nodes carry information about the domain plan

context using two distinguished arguments in each PS plan header, the first containing the name of the domain plan class being considered by that PS plan class and the second the list of arguments to that domain plan class. A metaplan like ***build-subplan*** can thus model the consideration by the agent of a subset of the current plan class by using in its subcalls the domain plan and argument values for a subclass of the plan class whose header was passed in as its first argument, while a metaplan like ***add-boolean-constraint*** that does not involve a change in domain plan context just passes to its children the same context arguments it received. For convenience, this context mechanism is also carried through metaplans like query plans that are leaves of the metaplan tree, so that it is possible from any node in the metaplan tree to access the relevant domain plan context.

The metaplan tree form offers many advantages as a model of problem-solving context. First and most important, it allows the representation of multiple possible domain plan classes for the same goal, for modeling contexts where the agent is in the process of comparing different possible solutions. The different plan subclasses of a single plan class, for example, become siblings in the metaplan tree. Second, the metaplan tree allows a unified representational form to capture both moves that further refine or specify the domain plan structure under consideration and moves that request data or that inform the expert about the agent's plans without modifying them. This single representational form makes it possible to apply a uniform style of heuristic rules to control search in the tree for related contexts. Third, the metaplan tree can represent a particular problem-solving context, information that goes beyond the domain plan context, but in a way that is integrated with the domain plan model, instead of requiring a separate structure cross-indexed to it. For example, the metaplan context can reveal whether the agent's current purpose in exploring this branch of the domain plan tree is to determine its feasibility or to evaluate its cost, which is an important distinction because the kinds of queries predicted in the two contexts are different. This kind of metaplan tree where each metaplan node references the domain plan context seems the simplest pragmatic modeling structure offering these advantages.

6.3.2 Four Classes of Metaplans

The PS metaplans fill four main functions in the model, each of which is implemented by a class of related metaplans:

- they outline the agent's exploration of domain plan subclasses and subactions and her variable instantiation moves that are part of searching for and refining a feasible plan for the goal,
- they predict what queries might be motivated by the exploration of different plan possibilities,
- they model alternative exploration patterns of the domain plan space that are based on evaluating relative cost rather than determining feasibility, and
- they allow for instances where the agent chooses to directly inform the expert about the plan path being explored, in order to establish the proper context for queries that might otherwise be misleading.

Metaplans of the class that model the agent's search for and refining of a feasible plan for the goal are called plan-building plans. The class of plan-building metaplans includes some that model the elaboration of the basic domain plan possibilities for the overall goal in terms of subplans and subactions, and others that model plan-building moves that instantiate open variables within the domain plans. These plan-building metaplans thus form a backbone for the metaplan tree in terms of the subplans, subactions, and variables of the domain plan classes; the current domain plan context at any node can be derived from the plan-building ancestors of that node, which give both the subplan class restrictions and the variable instantiations that characterize the current plan for the node.

In addition to the actual plan-building moves, the problem-solving agent also performs actions directed toward gaining the knowledge needed to choose between domain plan possibilities or to determine their likelihood of success. In Wilensky's terms, plan feasibility tests of this sort were part of his **simulate-plan** metaplan, determining whether those actions, if performed, would achieve the desired result. That determination is modeled here, of course, as a question about a precondition of the plan. The metaplans in this class are called the query plans, and the queries they model are the primary observable behavior in this discourse setting.

A third class of metaplans are evaluative in nature, and have to do with comparing the costs and benefits of different plans. These metaplans model a different kind of exploration of the same space of subplans and subactions from that modeled by the plan-building metaplans.

Finally, there is also a class of metaplans that model problem-solving actions on the agent's part that do not have any direct effect on the intended plan or on the agent's own knowledge, but that serve to inform the expert to whom the queries are directed of some fact about the agent's plan-building activity that the agent supposes the expert would not be able to discern from the sequence of queries alone, but that it is important that the expert should know in order for her to be able to track the agent's plan-building correctly and thus be as cooperative as possible. The informing metaplans model these explanatory actions on the agent's part.

6.4 The Plan-Building Metaplans

In this section, we describe those metaplans that the agent uses to refine and instantiate the domain plan structure that serves as the proposed solution to the problem. In our planning model, the elaboration of a plan for a given goal can be divided into two classes of tasks, first selection of one of the possible subplans at each node where there are multiple options, and second the instantiation of the open slots in the plans to values from the domain that satisfy the preconditions of the plan tree. We will likewise consider two different classes of plan-building metaplans, those that select from among the alternative plans at each choice point, and those involved in finding fillers for the variable slots in the plan tree.

Note that while the agent's aim is to construct a tree of plans and subactions with a single instance selected from the plan class for each goal, the development of that tree proceeds by exploration of alternatives, so that the metaplan tree model of the

agent's problem-solving process often has multiple active subplans for a single goal and multiple subtrees beneath a given goal exploring various instantiations for the variables. These plan-building metaplans can have the effect, therefore, either of creating initial subtrees to meet given goals or of adding additional subtrees to ones already considered to explore further alternatives, using either other subplans or alternative variable instantiations.

Note that while the refined and instantiated domain plan resulting from these plan-building metaplans is central to the agent's problem-solving goals, the actual choices modeled by these metaplans must usually be inferred by the expert from the queries asked, rather than being explicitly communicated. For example, a student does not usually say

> *I'm a B.S. student, CIS major, considering earning credit in CIS 360 this semester to help satisfy my core requirements for the degree. Is there space available?*

Instead of describing in that way each of the plan-building choices involved in specifying the plan, the agent in an expert advising setting simply asks the query

> *Is there space in CIS 360?*

expecting the expert to be able to supply from the query and from context a sufficient model of the motivating plan to enable helpful responses. In those cases where the agent expects that the expert will not be able to infer the plan correctly, she can use explicit inform statements to fill in the plan-building context, as we will see in Section 6.7, but that context is typically derived from the agent's query behavior. Thus, while these plan-building metaplans are at the heart of the agent's effort, they are seldom the explicit target of communication from agent to expert. Instead, they are deduced by the expert as having been executed by the agent when the agent goes on to perform some more explicit behavior based on the plan exploration embodied in the plan-building actions.

6.4.1 Metaplans that Outline the Domain Tree

One class of plan-building metaplans are those that model the refinement of the domain plan tree of subplans and subactions. As that process is formulated here, there are three such plans, one called ***build-plan*** that effectively encodes the domain plan nodes themselves, and two more, ***build-subplan*** and ***build-subaction***, that encode the links between nodes in the domain plan tree.

6.4.1.1 Build-Plan

This metaplan is the basic metaplan class encoding all the agent's problem-solving activity to build a domain plan for a particular goal, that goal being expressed by the header of that class of domain-level plans that share the given goal as one of their effects. All the rest of the metaplans considered here are subplans and actions of this ***build-plan*** metaplan, containing in more detail the various problem-solving approaches which the agent can use to achieve the goal of the ***build-plan***, which is to have a fully refined and instantiated plan of domain actions that will achieve the

desired domain goal.

To specify an instance of the ***build-plan*** metaplan, one must specify the domain goal for which the agent desires to create a plan. Given the structure of domain plans and plan classes, where a domain plan class represents all of the domain plans that share a given set of effects, such domain plan classes serve as a natural encoding for the domain goals that are the focus of the plan-building activity captured in the ***build-plan*** metaplan. Therefore, the arguments to ***build-plan*** include the domain plan header of the given domain plan class and whatever argument values are needed to instantiate its input variables and thus determine the problem toward which the plan building will be addressed.

For example, in the naval domain, suppose that one vessel in a particular battle group has experienced a breakdown that puts it out of action, and that the agent's domain goal is to restore the functionality (or "readiness" in Navy terms) of that battle group. On the domain plan level, there is a domain plan class for plans that achieve the goal, whose header is shown in Figure 6.2.

```
(increase-battle-group-readiness
     ?battle-group
     ?old-rdy
     ?new-rdy)
```

Figure 6.2: Plan Class Header for Increase-Readiness

In the metaplan tree, a ***build-plan*** node representing the class of problem-solving metaplans that can be used to build a plan for that goal could be expressed as in Figure 6.3.

```
(build-plan
     plan-header (increase-battle-group-readiness
                    ?battle-group ?old-rdy ?new-rdy)
     arg-values (?battle-group ENTERPRISE-GROUP
                 ?old-rdy C3
                 ?new-rdy C1))
```

Figure 6.3: *Build Plan* Node for Increase-Readiness

As mentioned before, the arguments to metaplans like this ***build-plan*** include the plan header and argument values for the domain plan class being considered. The metaplan action and subplan children of this node describe the problem-solving actions that the agent could pursue in order to further specify which domain plan subclasses the agent intends to select and to specify plans for the actions required in those plan classes or to achieve their preconditions.

While the metaplan context includes a full specification for the domain plan class that is the goal, it does not contain a full description of the world state in which the plan will have to function. While one could have included a world-state argument with the metaplans, we have chosen to allow that information to be present implicitly through the world model, since we are assuming that the world model in this setting is stable, given that the agent is currently building plans for future execution and that those plans are non-interacting. Like the world state, another important factor in the

expansion of a ***build-plan*** tree that is not included in the argument list is a model of the extent of the agent's knowledge about thee current situation, since many of the plan-building actions have to do with gathering pieces of information about the world context and applying them to tne choice of how to extend the domain plan tree to meet the desired goals. We discuss in Section 7.2.2 how the influence of the agent's world knowledge can be modeled as part of the heuristic component that governs plan tree growth.

6.4.1.2 Build-Subplan

One subset of the set of possible metaplans modeled by a ***build-plan*** node can be formed by restricting the domain plan class to one of the particular domain plan subclasses of that plan class. For example, underneath **increase-battle-group-readiness** we might have domain plan subclasses for **add-ship-to-group**, **replace-damaged-ship**, and **repair-damaged-ship**. Each such domain plan subclass could serve as the basis for a ***build-subplan*** metaplan, modeling the agent's intention to explore a plan for the goal that falls within the classification of that particular subplan. A ***build-subplan*** node thus represents a choice by the agent at least temporarily to focus the space of plans being explored to those that fit within the chosen subplan classification.

The arguments of ***build-subplan*** include both the specification of the higher-level domain plan class and its arguments that is standard for all the metaplans and the header of the particular domain plan subclass that this instance of ***build-subplan*** selects. In the current model, the only direct subplan of ***build-subplan***, in turn, is a ***build-plan*** node for that domain subplan, with its argument values updated properly to represent the situation of that subplan in the given plan context. However, one extension that was considered during development was a richer set of metaplans encoding the various means the agent might use to select one particular subplan class. For example, ***use-default-subplan*** might be a more specific version of ***build-subplan*** that would apply in cases where one of the domain plan subclasses is used much more frequently than others. In Pragma, that distinction about typical usage patterns for plan subclasses was included eventually in the heuristic component, rather than as a separate metaplan.

6.4.1.3 Build-Subaction

In the same way, the agent may choose to explore the plans that would be required to fulfill one of the subactions of the domain plan class referenced by the parent ***build-plan*** node. Remember that the subactions attached to a plan class are those belonging to all the members of the class, so that achieving the goal by means of any plan in the class will require also selecting a feasible plan for the subaction, and the ***build-subaction*** metaplan models the problem-solving step of beginning consideration of that issue. For example, every plan in the **earn-credit-in-course** class might include a **pay-semester-bill** action, so that that action would be attached at that level of the plan class hierarchy. Then one of the problem-solving steps open to an agent pursuing a ***build-plan*** for **(earn-credit-in-course CIS360 89-A)** would be a ***build-subaction*** for **pay-semester-bill** which would in turn spawn a ***build-plan***

for that plan class that might in turn be expanded by *build-subplan* into the alternatives pay-cashier and mail-check.

As with *build-subplan*, the arguments to *build-subaction* include both the domain plan class header and argument values inherited from the parent *build-plan* and also the header specifying which subaction of that plan class is to be explored.

Given just the three plan-building metaplans presented so far, we can see that there will be a one-to-one correspondence between the skeleton of a metaplan tree based on just these plan-building metaplans and the domain plan tree that could have been built directly from the domain plans with their subplans and actions. The main advantage of modeling this structure by means of plan-building metaplans is that they form a foundation for attaching the metaplans that model the other classes of problem-solving steps. They also establish a base for further expansion in that the plan-building metaplans themselves can be further refined, as pointed out in the previous section about *build-subplan*.

6.4.2 Metaplans that Fill Variables

In order to achieve a fully-specified plan for the domain goal, the agent needs not only to select plans for the subplans and actions, but also to instantiate any open variables in those plans to actual entities in the world model that fulfill the preconditions of those plans. This is because even a full choice of subplans and actions does not specify a unique plan, but only a template that still requires the instantiation of its variables to specific values in order to produce a complete plan. The metaplans in this section capture this process of selecting values to fill the variable slots in the plans.

6.4.2.1 Instantiate-Var

The *instantiate-var* metaplan describes the problem-solving step of binding an open variable within a domain plan to a particular value. For example, in the naval domain, the plan for replacing one ship by another introduces a variable, ?rship, standing for the ship to be used as the replacement. The effect of an *instantiate-var* metaplan would be to create a context in which that variable was bound to the proposed value. If the agent asks how long it would take the Fox, say, to reach the station of the damaged Sterett, the expert recognizes that the agent is considering the possibility of replacing the Sterett with the Fox, which is modeled here by an instance of the *instantiate-var* metaplan appearing beneath the *build-plan* for replace-ship. The effect of *instantiate-var* is to create a new, alternative context in which the chosen variable is bound to that given value. Just as *build-subplan* can model consideration of various different plan subclasses, *instantiate-var* also does not cut off consideration of alternative plans. The metaplan context is powerful enough for parallel domain plan contexts to be explored in parallel.

The occurrence of an *instantiate-var*, like that of *build-subplan* and the other plan-building metaplans, is not usually expressed directly in the dialogue, but

is observed indirectly by deducing the metaplan steps that could motivate the observed query behavior. The expert goes through a process of deducing the metaplan moves necessary to create the context from which the query could be asked. Thus, in the above example, the query implies the existence of some context that would motivate a query about the distance from the Fox to the Sterett, and the expert is able to deduce given the previous context and knowledge of the domain plan possibilities the implication that the agent is considering a **replace-ship** plan with the Fox as the **?rship**. This process of deduction, from a given context and query, of the possible metaplan contexts that could have motivated the query is central to being able to maintain and make use of such models.

Instantiate-var is a metaplan class with some useful subclass structure which models particular kinds of variable instantiation that the Pragma model wishes to distinguish. There are subclasses of ***instantiate-var*** that can be used when it is clear why the agent has chosen a particular instantiation, for instance, if there is only one value possible, or if the value chosen is one included in the agent's query. For example, ***choose-only-possible*** is a subclass of ***instantiate-var*** that applies when only one of the choices to fill a slot causes the preconditions of the plan to be satisfied. Two other subclasses of ***instantiate-var***, ***pick-value-suggested*** and ***pick-at-random***, distinguish in the search for a match to a query whether the possible instantiation value was mentioned in the query or not; the case where a particular value is suggested can be heuristically favored, since it does not expand the tree as greatly as the random version that builds instantiation nodes for every possible entity of the given class.

Subclasses like ***choose-only-possible*** tell us more about the agent's choice process and it's motivation, which can be useful heuristically in predicting the likely future course of the dialogue. However, there is frequently no data for more exact predictions of this sort, especially because modeling the reasons for the agent's choice of particular values can depend on more knowledge about the agent's beliefs than is available in the expert advising setting. For example, a student who needs one more elective in the sciences to fulfill the distribution requirements might ask about the meeting time of BIO-110. While that question does enable us to conclude that the agent is considering enrolling in that course, and thus that the agent's domain plan tree has been extended by instantiating the variable **?course** in the **earn-credit** plan to BIO-110, we do not have enough data to determine whether this variable instantiating metaplanning step in this case resulted from, say, knowledge that that was the only course whose schedule matched the available time slots, or whether the student was following up the recommendation of a friend, or whatever in this case made the agent choose that particular value. Thus ***instantiate-var*** by itself encodes only the plain fact that the domain plan has been extended by the instantiation of that variable to that value; the more precise characterizations in the metaplan subclasses of ***instantiate-var*** are available for cases where sufficient information is available.

6.4.2.2 Constrain-Var Plans in General

While direct instantiation of an open variable to a particular value is the usual approach, there are circumstances where agents make use of intermediate stages that restrict the possible values without specifying any particular value, and these are modeled by ***constrain-var*** metaplans. The effect of these metaplans is to restrict the space of possibilities from which a value for the variable must be chosen. Examples of such restrictions include those underlying queries in the damaged ship context like

List the cruisers in the Indian Ocean.

or

List the cruisers within 500 miles of Sterett.

where the space of candidates to be considered is restricted by the query. While the restriction appears as part of the query, it is modeled as a metaplan step modifying the actual partial plan under consideration, since the query reveals a narrowing of focus on the agent's part at least for primary consideration of possible fillers.

The representation of these constraints depends on the introduction in variable instantiation query contexts of a new variable standing for the set of possible fillers of the open variable. A variable of this sort is generated in the metaplan tree at the ***build-plan*** nodes for each free variable in domain plans that introduce them. When such set variables occur in normal atomic formulas, the interpretation is that the constraint of the formula is applied distributively to each member of the set, so that the constraints applied to the free variable in the plan preconditions are applied in the metaplan tree to the derived set variable and thus to each member of the possible fillers set. An existing plan constraint like **(readiness ?rship C1)** then becomes **(readiness ?rship-set C1)**, encoding that each member of the set of possible fillers must be in the top readiness condition.

A single query can reveal the agent's application of a number of such constraints on the set of possible fillers, as in the example

List the unassigned C1 frigates.

This use of multiple ***constrain-var*** nodes on a single branch raises the danger of duplicate branches in the metaplan tree that differ only in the order of application of such a set of constraints; to avoid this, the system imposes an arbitrary (alphabetical) order on such sequences of ***constrain-var*** nodes. Alternatively, the agent can add further constraints to the set identified in a previous query, producing a sequence of increasingly specific subsets like the following:

List the unassigned ships.
Which of them are frigates?
Which of those are C1?

For such examples, new branches that reflect the added constraints are sprouted from the branch modeling the previous query.

Note that there are two classes of constraints that can be involved in ***constrain-var*** metaplans, either constraints that are already present as preconditions

in the referenced domain plans or constraints that the agent adds on her own. Added constraints that are not also preconditions must be either factors that the agent independently knows will affect performance in the current circumstances, although they are not required for plan success, or else restrictions that the agent has other reasons for wishing to apply not derivable from the current plan context. An example of the former would be where the agent restricts the set of replacement ships to be considered from all frigates to frigates within 500 miles, supposing that there are some within that range and that they will be able to get to the station more quickly, although any frigate would be able to fill the slot successfully. An example of the latter would be where the agent excludes frigates in the Indian Ocean from consideration because of independent knowledge that there will soon be a mission there requiring them all, or because she is also considering initiating an exercise for all the units in that area. In the former case, the one we are mainly concerned with, while the system cannot predict effectively the exact restriction the agent may use, it is possible for the system to connect the restriction with preconditions in the plans, demonstrating its relevance. Thus, while the pick of *500 miles* is arbitrary, the distance between the replacement ship and the damaged one can be seen in the plans to be relevant because it is the distance which the replacement ship must cover in the **sail** subaction to take up its station. However, there is no way for the system to judge the relevance of the latter sort of added constraint, since that relevance depends on data known only to the agent; the ***constrain-var*** metaplans still allow for such restrictions, but with a lower heuristic ranking.

In theory, another unusual sort of constraint that must be allowed for is a random one, where there is no motivation at all, not even a hidden one, for the specific restriction except that it will serve to cut a large candidate set down to a size that is more efficient to pick from. For example, if someone advertised a position on the help wanted pages and received hundreds of responses, a reasonable strategy might be to select one tenth of the candidates at random, making it possible to read the applications in a reasonable time while still leaving a large enough pool to choose from intelligently. This unusual sort of restriction would naturally also be one which the system could not connect to the plan preconditions.

This ability of ***constrain-var*** metaplans to add unrelated constraints to the tree context creates a danger of unintentional matches. The heuristic weight given to the addition of such novel constraints must be severely limited to prevent queries from linking to nodes from locations in the tree that are completely unrelated to the real plan context, merely by building a sequence of ***constrain-var*** nodes that end up matching the entire query. In order to prevent such false matches, it is important to direct the search so that any natural match to a relevant section of the tree is found before a match that requires addition of one or more novel constraints. We will see later how heuristics based on factors like the importance of the constraint to the task, the size of the unconstrained set, the predicted likelihood of members of the unconstrained set meeting the constraint, and the likelihood of the agent knowing already whether members of the set meet the constraint can be used to control the addition of these variable-constraining metaplans and thus to focus the search on the more likely derivations.

6.4.2.3 Add-Boolean-Constraint

The actual metaplans for adding constraints are separated into plans for adding boolean constraints and for adding scalar constraints, since the form and heuristics governing the two are somewhat different. Adding boolean constraints is one way of dealing with underconstrained variable choices, where there are a large number of possible fillers, by adding explicit constraints to those implied by the plan context itself. Using a naval example again, the agent may ask for a list of the fast frigates (a separate subclass) when the vessel needed for replacing the damaged one according to the plan constraints could be any sort of frigate. Asking for a list of frigates *within 500 miles of Sterett* would also be classed as a boolean constraint, although more than one atomic sentence is used to express that constraint, as in Figure 6.4.

```
(distance-between ?damaged-ship-loc ?rship-loc ?dist)
(<= ?dist 500)
```

Figure 6.4: A Multiple-Assertion Constraint

The ***add-boolean-constraint*** metaplan takes only the normal plan context arguments, since the constraint being applied appears as a free variable of this metaplan. The algorithm for searching the tree ensures that ***add-boolean-constraint*** nodes are only built for constraints that do appear in the query, and the heuristics try to ensure that related plan contexts in which the constraint already appears as a precondition will be found in the search before contexts to which it has been added by this metaplan.

6.4.2.4 Add-Scalar-Constraint

Add-scalar-constraint is a metaplan that restricts the fillers of a set to those that maximize or minimize the value of a particular scalar function, or that fall within a given percentile on that function when compared to all the possible candidates. This metaplan specifies both the function to be maximized and the ordinal ranking of the resulting values to be permitted, that is, whether to accept only the best value on that scale, the best N, or any value in the top N%. An example in the naval domain where the agent is involved in filling the **?rship** slot would be adding a constraint that restricts the possible fillers to the five vessels closest to the damaged ship, or to the single vessel most recently overhauled.

For representing scalar constraints, the special **first-n** or **first-n%** forms are used, which define the restricted set according to the scalar function and cutoff values. For example, the restriction to the five closest vessels is phrased as in Figure 6.5. This clearly begins to stretch the bounds of the logical representation using sets of atomic formulas; a more powerful logical form would be required to represent more complicated added constraints, but that would also complicate the test for matches, and it is not central to the general metaplan modeling approach.

It is interesting to compare scalar constraints and boolean constraints in terms of their problem-solving purpose. Both are used to restrict the space of possible fillers that must be considered, typically because the agent expects there to be many more

```
(first-n 5
   ?rship-set
   ?dist
   (distance-between ?rship-set Sterett-loc ?dist)
   increasing
   ?restricted-rship-set)
```

Figure 6.5: Assertion Restricting Set to Five Closest Vessels

candidates than necessary. Scalar constraints serve that limiting purpose directly, by specifying an absolute or relative size for the desired subset along with the criterion to be used to select it. They are subject to failures only if the entire set of possible fillers is empty, or less than the absolute set size stated. Boolean constraints, on the other hand, can fail when used in this way in spite of an adequate pool of possible fillers if the fixed constraint happens to be strict enough to leave only the empty set.

6.5 Query Metaplans

The query metaplans, the second major class of metaplans, are a class of problem-solving moves whose effects do not directly change the plan the agent is considering, but instead provide the agent with information useful in evaluating the various choices of subplans or variable instantiations. They are divided here into the two subclasses of plan feasibility queries, which are used to determine whether the preconditions of a particular plan class will allow it to succeed or not, and slot data queries, which are used to gather information about the possible fillers of a free variable slot for use in instantiating it.

6.5.1 Plan Feasibility Queries

In the course of planning an approach to the goal, the agent may consider many different possible plans, and different instantiations of the variables in those plans. Plan feasibility queries, the most straightforward of the query metaplans, are queries whose goal is to help determine whether or not a particular domain plan or instantiation will be executable. This is done by examining the values of the plan preconditions for the given variable bindings, knowing that a plan whose preconditions fail will not be executable.

The space of plan feasibility queries representable in the metaplan tree depends on whether or not the plan preconditions of each node reached are tested against the database to prevent building structure for infeasible plans. As discussed in Section 8.5, Pragma can be run either in an instantiation mode or in a mode where preconditions are not tested against the database. The instantiation mode would be the natural choice if our goal was efficient planning, since that mode guarantees that any plan actually found in the tree is feasible (up to the accuracy of the system's world model). However, since our interest is in modeling the agent's planning process in order to respond cooperatively and handle ill-formed input, it is important here that we be able to include in the plan tree plans that are actually not feasible, but that the agent might consider to be possibilities either through not knowing the facts that block them or through not having worked out those implications. Plan feasibility queries

concerning such plans can be recognized only when Pragma is running in this non-instantiation mode. (A fuller comparison of the advantages of each mode can be found in Section 9.5.1.)

6.5.1.1 Ask-Pred-Value

The easiest way in this setting for the agent to test a precondition is simply to ask the expert what its value is for the given variable instantiations, and that is the planning action encoded in the ***ask-pred-value*** metaplan. In the **replace-ship** example, the agent might ask whether the proposed replacement ship is C1 or not, since that is a precondition of the plan to substitute it for the damaged one. In that example, the ***ask-pred-value*** metaplan node would include a free variable for the predicate being queried which could be instantiated to any of the plan preconditions introduced by the domain plan at the current node, including in this case (**readiness ?rship C1**). Thus the system's default behavior, unless particular heuristics come into play, is to explore beneath a particular ***build-plan*** node the possibility of queries about each of that plan class's preconditions.

Note that the restriction on ***ask-pred-value*** that it only query preconditions of the current domain plan class rather than those inherited from plan superclasses at higher nodes does not restrict the power of the model, since queries about those preconditions can be raised by ***ask-pred-value*** nodes attached at that higher level. The restriction merely serves to avoid redundancy in the metaplan tree, where the same query could be matched not only at the level of the domain plan class that introduces the precondition but also at every child plan class.

6.5.1.2 Check-Pred-Value

An alternative query form for determining the value of a plan precondition if either the required value or a likely value is known is to ask a yes/no question, modeled by the ***check-pred-value*** metaplan, distinct from the preceding because the form of the query involved is different. In our current example, that corresponds to asking

Is the Fox C1?

rather than

What is the readiness of the Fox?

since the domain plan requires that **?rship** have the top readiness value.

While the typical use of ***check-pred-value*** is to ask whether the actual value matches the one required in the plan preconditions, the agent can also use it in a negative sense where a "yes" answer would mean the plan was not feasible if a particular value for the predicate is especially likely. For example, a student registering for the next semester and having a hard time finding courses with space available might ask

Is CIS 145 full, too?

as a way of testing the feasibility of the plan of taking that course, even though a "yes" answer to the query implies that the plan is not feasible.

6.5.2 Slot Data Queries

While plan feasibility queries are used in deciding which of several subplans to pursue, and therefore belong with the tree-outlining ***build-subplan*** and ***build-action*** metaplans, slot data queries, the other major class of query metaplans, are used to gather data about the possible values for the variable slots in the domain plans, and thus are used together with the ***instantiate-var*** and ***constrain-var*** plans. As always in our model, the agent is assumed to know the domain plans, and thus also the existence of the open variable slot and the plan constraints upon its filler, but not assumed to know what actual entities in the database are of the proper type to be possible fillers or actually to fit those constraints.

As pointed out in Section 6.4.2.4, the slot data queries are formulated in terms of the set of possible fillers for the slot, those entities in the world model of the specified type for the slot that are therefore conceivable fillers for the slot. That set can be restricted, as pointed out earlier, by using ***constrain-var*** metaplans with constraints either taken from the plan preconditions or added explicitly by the agent.

In this section, we present the metaplans for the different kinds of queries that agents can ask about such sets of possible fillers, both simple queries like whether they are non-empty, what their cardinality is, or for a list of their members, and also more complex queries, assembled from a number of slot data query metaplans, like for a list of members sorted on some relevant parameter or for only a certain number of the set's members.

6.5.2.1 Ask-Existence

The ***ask-existence*** metaplan models queries by the agent as to whether a given set of possible fillers is empty or not. For example, in the naval domain, the agent might ask,

Are there any unassigned cruisers?

A negative answer would quickly rule out the entire **replace-ship** plan, since it depends on the existence of a ship that can be used as the replacement. A positive answer, as far as it goes, would justify continued exploration of the plan. If the set used in the query is exactly the admissible set, that is, includes all the restrictions on the slot found in the plan context, then a positive answer demonstrates that satisfactory values for that slot can be found. However, if the query set is a superset of the admissible set, formed by omitting some of the constraints, then a positive answer still leaves open the question of whether a fully satisfactory value can be found. However, a negative answer for a superset, naturally, implies the same answer for the admissible set itself, and demonstrates that the plan is not instantiable.

Since the ***ask-existence*** query gives the agent only a proper subset of the information available from other types of queries like ***ask-fillers*** or even ***ask-**

cardinality*, one might ask why it would ever be used. In logical terms, the only justification seems to be if the effort involved in answering or in absorbing the answer to the existence query is less than that required for the more informative ones. The increased ease of answering and absorbing might be predicted to balance the decreased information content, and the existence query can always be followed by a more informative one. There are also other layers of constraints on the discourse that might encourage the choice of this metaplan, since the agent, as a cooperative communicant, is constrained by Griceian principles not to ask for more information than is necessary to the purpose or for more than will be useful. Thus, asking for a full listing of the candidates or even for the exact number of candidates when the list would be large and no effective use would be made of the further information would violate that cooperative principle. Use of these weaker slot data query metaplans may be justified on those grounds.

6.5.2.2 Ask-Cardinality

The ***ask-cardinality*** metaplan goes one step beyond ***ask-existence***, querying how many entities of the given type meet some or all of the requirements attached to the slot in question, for example,

How many C1 frigates are there?

This information about the number of fillers in the set, in the case of a non-zero answer, provides in addition the basic existence test data useful for selecting a metaplan strategy to determine a desirable instantiation. Obviously, if the cardinality of the required set is 1, then no further questions need to be asked, and further planning and testing can then assume the use of the single feasible instantiation for this variable slot.

Even when the cardinality is greater than , the size of the set may help determine which metaplanning strategies will be most useful. A small set (say, less than 10) can be listed in full, while a large set suggests the addition of further constraints, related to performance if possible but random if necessary, to produce a set from which a choice can reasonably be made. The fact that the useful value of the response to an ***ask-cardinality*** query for further metaplanning is based not so much on the exact value of the cardinality as on whether it is *0*, *1*, *few*, or *many* explains the informal cardinality queries that occur, like

Are there many deployable cruisers?

that only ask for enough information about the size of the candidate set to be able to select appropriate metaplans for continuing the plan building. The portion of the heuristic component that depends on the agent's world knowledge can help to indicate situations where the agent knows very little about the possible filler set and thus is relatively likely to use an ***ask-cardinality*** metaplan, and can also make use of the added knowledge provided by the system's answer to adjust the predicted scores for different follow-on metaplan steps.

6.5.2.3 Check-Cardinality

The *check-cardinality* plan is similar to *ask-cardinality* in purpose and setting, except that the actual query form is different, since the agent asks only the yes/no question of whether the cardinality of the set of possible fillers is equal to or greater or less than a particular value, for example,

Are there two other carriers in the Indian Ocean?
Are there more than 5 C1 destroyers?

As with *check-pred-value*, this is the weaker plan of the pair, but it can be justified in particular discourse contexts. The equal test, for example, suggests that the agent is already fairly sure of the answer, and merely wants to confirm that the expert shares this understanding, while the greater or less than tests suggest that the queried set size is a logical divide between two different follow-on query strategies. *Check-cardinality* queries can also be asked in a negative form, as follows:

There aren't more than 2 free carriers, are there?
No places are open in CIS 214 or 216, I suppose.

In those cases, too, the justification seems to be that the limit being tested is one of the determining factors for a particular path being taken in the agent's planning process. By drawing attention to that factor in the process of confirming it, this sort of query can also inform the expert about the choice and its motivation, so that the expert's model of the agent's plan can be updated; the expert can then either confirm the agent's facts and deduction, perhaps confirm the facts but not the deduction by pointing out alternative paths to the given goal in the current context, or else inform the agent that the facts supporting the choice are not or are no longer true.

6.5.2.4 Ask-Fillers

The *ask-fillers* metaplan is the basic plan allowing the agent to learn which entities in the world model are candidates for a particular slot in the domain plan under consideration. For example, our naval planner might ask

Which cruisers are C1?

or a student registering for courses might ask

Which 300-level Computer Science courses have space available?

The restrictions applied to the goal set in the *ask-fillers* query can either be inherited from plan preconditions in the current context or they can be added constraints that are not already present in the tree; in either case, the query is matched by first building appropriate *constrain-var* nodes to match the additional constraints and then attaching the *ask-fillers* node.

Clearly, an *ask-fillers* query returns a superset of the information returned by either *ask-existence* or *ask-cardinality*, unless the answer is "None", in which case all three return exactly equivalent information. The difference between the amount of data required for an answer to the different queries increases substantially with the size of the set, so that, in terms of Grice's maxim of relevance [21], an

ask-fillers query is inappropriate if the size of the resulting set is known to be very large.

In addition to the further plan constraints that can be introduced by *constrain-var* moves, *ask-fillers* plans can also be modified by one or more of the following metaplans, *limit-cardinality*, *sort-set-by-pred*, and *request-attribute-value*. These query modifying metaplans change the type, amount, or format of the data to be displayed about the identified set of possible slot fillers. Unlike the plan-building *constrain-var* plans *add-boolean-constraint* and *add-scalar-constraint*, these query set constraining plans affect only the results of the query and their interpretability, but have no direct effect on the value that will be chosen as the instantiated value for the open variable. In the metaplan tree model, the nodes for these query modifications appear as children of the *build-plan* node introducing the open variable and parents in turn of the *ask-fillers* node that models the modified query. The model allows more than one of the modifier metaplans to be applied on the same *ask-fillers* branch in order to handle queries like

List the C1 destroyers sorted by speed with their locations.

There are structural constraints on the possible modifier combinations, like that only one instance of *limit-cardinality* makes sense in a single *ask-fillers* branch. These constraints are not yet captured in the model, however, since the search routine only instantiates modifier metaplan nodes when the query form suggests them, so that overgeneration of inconsistent combinations is avoided in that way.

6.5.2.5 Limit-Cardinality

The *limit-cardinality* plan is a modifier on *ask-fillers* that allows the agent to place a limit on the number of possibilities returned. That limit can be either explicit, as in the query,

Name five of the courses that satisfy the humanities distribution requirement.

or approximate, leaving the expert some discretion, as in

What are some of the courses that satisfy...

The explicit form is not often used, since an agent involved in an *ask-fillers* metaplan usually does not have enough information to motivate an exact limit. (That kind of query is used in exams, of course, where the professor does know the distribution of the candidate set, and is therefore not interested in being informed by the answer, but in judging the knowledge of the question answerer, but that discourse setting is quite different from expert advising.) The informal version of *limit-cardinality* serves the purpose of allowing the expert to return a partial list where that would be as informative as the complete one, given the agent's intended use of the list as a source for plan variable slot instantiations.

This plan shares some of the flavor of *add-scalar-constraint*, which the agent can use to limit the size of the set returned. It differs from it by not specifying at all how the set size is to be kept down, allowing even random subset selection.

6.5.2.6 Sort-Set-by-Pred

The *sort-set-by-pred* metaplan allows the agent to piggyback along with a request for a list of candidates to fill a variable slot a request for information about the ranking of the candidates along some scale. For example, our naval agent might ask

List the C1 cruisers in order of decreasing speed.

or

... in order of increasing distance from the damaged vessel.

The scale used for sorting is typically one that influences the marginal cost of the plan, and that thus would be a way of choosing from among the list of candidates provided. If there is a fixed limit on the scale value in the plan preconditions, such a limit would usually be included on Gricean grounds directly in the query, as in

List the cruisers whose readiness is at least C3, in order of readiness.

to avoid asking for detailed information that could not possibly be of use, since it serves only to distinguish between different classes of impossible candidates.

Specification of a *sort-set-by-pred* metaplan in an *ask-fillers* context requires specification both of the scalar function by which the ranking is to be done and of the direction along that scale which is to be ranked more desirable. The function used for the scaling may be one already referenced in the plan, like the readiness of a vessel, one derived from information present in the plan, like the distance of the possible replacements from the damaged vessel as calculated from their locations or perhaps the time required for them to reach that position based on location and speed, or, less probably, scaling may, like an additional constraint, be based on some other attribute of the candidates whose relevance is not derivable from the plan context model, since it depends on factors known to the agent and not the expert. All of these are modeled as instances of *sort-set-by-pred*, but the heuristic component judges the former where the scalar predicate is already referenced in the plan context as more probable.

6.5.2.7 Ask-Attribute-Values

The final slot data query modifying metaplan, *ask-attribute-value*, adds to the query by requesting information about the values of some attribute or attributes for each member of the candidate set. For example, our naval agent might ask,

What are the locations of the C1 cruisers?

The goal of this metaplan is presumably to use the attribute values as a way of choosing a candidate to instantiate the slot. It can either be done, as in that example, in a single query that identifies both the desired set and the attributes(s) of interest, or else the attribute value question can build on an existing slot data query that has already identified a candidate set. The same issues apply in this case as in the previous metaplan as to the selection of the specific attribute values to be displayed, that they are most often attributes whose relevance is already clear by their being present in the

plan tree preconditions or calculable based on such attributes, but that there is also the less likely possibility of the agent asking for values whose relevance can not be determined from the plan model.

The choice of which of the modifications to *ask-fillers* is most appropriate for a given combination of plan context constraints and agent world knowledge depends on the comparative cost of the different types of queries, which varies given the sorts of mechanisms used, both for storing and searching the database and for formulating and returning the answer. Thus, ***ask-attribute-value*** queries for the complete information about an attribute are more likely with computer database systems, where the additional cost of retrieving and displaying the attribute information is small compared to the fixed cost of formulating and entering a query, than they are with a human expert, where those cost ratios are inverted. In Griceian terms, again, the extra work required of a human expert to return a full table for the agent's use is seldom justifiable under the principle of not asking for more information than required, while in dealing with a computer system, the small marginal cost of also printing out the attribute values (especially when the query has already required that those records be referenced) may easily be thought worth it just on the chance that the extra information provided might prove useful. Another factor in the choice is the agent's view of the cooperativeness of the expert, whether she can be trusted to expand on the literal answers if there are marked patterns in the data that would be useful in selecting an instantiation, for example, answering a request for the two fastest cruisers by saying

> *Wilson, Frederick, and Vincennes can all do better than 28, while the rest are much slower.*

or perhaps to point out a misleading use of a scalar ranking, as when the "fastest" ship is only a tiny amount faster than another group of possibilities. The Pragma model is not currently refined enough to make use of these kinds of estimated cost factors in predicting the form of *ask-fillers* queries.

These modifier metaplans in combination with *ask-fillers* and the ***constrain-var*** plans capture a wide variety of the query behavior agents can use to gather information about possible fillers for free variable slots while building plans.

6.6 Evaluative Metaplans

The plan-building and query classes of metaplans are the most developed part of the model, and represent the behavior of an agent who is searching for a feasible plan within a tree of possible plans whose precondition requirements the agent knows but where the world state and therefore the actual success or failure of those preconditions for given instantiations are not known. This section on evaluative metaplans introduces a separate class of metaplans that are not concerned with identifying a feasible choice, but with calculating the cost of a particular plan and comparing that with the cost of alternative plans. For example, if our naval agent asks about the Fox's fuel consumption rate, that query is not relevant to the feasibility of using it as the replacement vessel, given the preconditions of **replace-ship**. It could be matched by an ***ask-pred-value*** plan using a predicate not included in the plan preconditions, but the context then would not capture the distinct flavor of the evaluative purpose of that plan. A more precise model results from attaching such

queries to a separate branch of evaluative metaplans.

The advantage of separating out evaluative metaplans comes in better model prediction of related follow-on queries. This is partly because evaluative queries often do concern different features of the plan context than feasibility queries, so that an agent whom the metaplan context indicates is evaluating the cost of a particular plan is likely to ask follow-on queries that are also evaluative, since the current metaplan context is concerned with the cost of the plan, assuming its feasibility. The other main advantage of separating out the evaluative metaplans is that it allows the heuristic component to take account of the different patterns of movement within the metaplan tree that are appropriate in evaluative rather than plan feasibility contexts. In plan feasibility contexts, the usual pattern embodied in the heuristics is a depth-first exploration of each plan branch, continuing the exploration of that branch as long as the query answers still suggest that it is feasible. In evaluative contexts, on the other hand, the agent is typically comparing the costs of alternative plans, so that movements in the plan tree to evaluative contexts attached to parallel plan branches become heuristically favored over deeper exploration within the current branch.

The evaluative metaplans are included in the model as a set of plan exploration metaplans parallel to the normal plan-building set of ***build-plan***, ***build-subplan***, and ***build-subaction***. An evaluative metaplan subtree can be attached as a child of any ***build-plan*** node, and the ***evaluate-plan***, ***evaluate-subplan***, and ***evaluate-subaction*** metaplans then form the foundation of an alternative exploration path over the same domain plans. Within the evaluative subtree, the normal domain plan feasibility constraints are not available for query matching, being replaced by a set of relevant evaluative functions stored with each domain plan. For example, the **replace-ship** plan carries as precondition restrictions that the class of the replacement ship be the same as the damaged one and that the replacement ship be in deployable condition itself, since otherwise it could not take over its mission. In contrast, factors like the distance the replacement ship would have to travel or the amount of fuel that would be consumed in doing so are not preconditions, but instead evaluative factors that the agent may well consult in deciding which plan to adopt but which have no direct effect on feasibility.

Note that the choice of whether a factor is coded as an absolute precondition or as an evaluative formula is to some extent arbitrary. While one could imagine using a destroyer from the Persian Gulf to replace a damaged one in Tokyo, for example, the evaluative functions of cost and delay would be so severe as to be prohibitive, so that that option might well be ruled out of **replace-ship** by a precondition test. The choice of which factors to model as preconditions is part of the definition of the domain plans as initially acquired, and the heuristic split between ***build-plan*** and ***evaluate-plan*** merely builds on those initial definitions. In the domain plans for Pragma, pairs of plans have sometimes been used to get around this problem, separating, for example, **sail-without-refueling** and **sail-with-refueling**, which has the effect of limiting the former to distances that can be sailed with fuel already on board, using that as a precondition for that limited plan, but as only an evaluative function for the unlimited one. Once such a separation is made, the heuristic system can take account of the differences between the two plans by means of the heuristics discussed in Section 7.2.3 that are sensitive to the raw probabilities of particular plans for a given goal, so that the unusual cases like replacing a damaged vessel with one from very far away will be

given appropriately lower heuristic weights than the more frequently used plans.

This parallel tree approach where the evaluative metaplans build their own tree structure is a useful heuristic model for the difference in metaplan context between plan feasibility queries and evaluative ones, although it does involve a duplication of domain plan structure that could perhaps be avoided in a reworking of the model using a higher-level metaplan for plan exploration that would trace out the domain plan structure and that could in turn be specialized to metaplans for either feasibility queries or evaluative ones. The difference between these two styles could then be made to flow naturally from their different evaluation of the comparative importance of the payoff spread between good and bad examples of feasible plans compared with the cost of the planning effort required to separate the two. The plan feasibility strategy assumes either that the differences between plans are not significant or that the effort of planning is great enough to outweigh any such differences, while the evaluative strategy assumes that the cost of planning will be justified by identifying a better plan. A higher-level metaplan that could capture that relationship between these two strategies and thus model more flexible combinations of the two would be a useful extension to this metaplan model.

The following subsections first present the basic ***evaluate-plan*** metaplan scheme and then discuss some possible higher-level metaplans that could be used to capture particular structures of plan evaluation that occur in problem solving.

6.6.1 Evaluate-Plan, Evaluate-Subplan, and Evaluate-Subaction

The ***evaluate-plan*** metaplan is the equivalent in the evaluative context of ***build-plan*** in the normal plan feasibility context. Because it is one of the children of ***build-plan***, it allows an evaluative subtree to be attached at any level of the plan tree where the heuristic search might call for it. The effect of ***evaluate-plan*** is to add not the preconditions of the current plan but its evaluative factors to the set available for matching to an input query. This approach covers both simple ***ask-pred-value*** evaluative queries like

What is Fox's speed?

and also more complex query constructions. A query like

What is the fuel consumption of the C1 frigates?

could be matched by first attaching an ***evaluate-plan*** node beneath the ***'·plan*** for **replace-ship** and then adding nodes for the query metaplans ***ask· ılue*** and ***ask-fillers***. Later queries might then also match to nodes of an evaluative subtree rooted in that ***evaluate-plan***.

Evaluate-plan is treated specially in the searching of the metaplan tree because is it unclear over what range of plans the evaluation is intended to apply. Thus, if the search moves above the original ***evaluate-plan*** where it attached to the ***build-plan***, the attachment point is also moved up one level to the next higher ***build-plan***, copying the intermediate structure and converting it into the parallel evaluative structure. For example, suppose the previous context was established by the query

What is Fox's speed?

which matched to an ***evaluate-plan*** node attached to the ***build-plan*** for the sail action. If the next query were

What is Perry's speed?

the evaluative subtree would be propagated upward, copying the structure up to the **replace-ship** level, yielding an evaluative subtree rooted in an ***evaluate-plan***, for **replace-ship.** This larger evaluative branch would be included in the search using the special evaluative heuristics that encourage comparative exploration at the same level, leading to a quicker match under the ***instantiate-var*** child of that upper ***evaluate-plan*** that instantiates the **?rship** to the Perry.

The other two metaplans in the ***evaluate-plan*** family are parallel to those that go with ***build-plan***, following the subplan and subaction links that outline the domain plan structure. The ***evaluate-subplan*** metaplan is the corollary of ***build-subplan***, allowing expansion of an evaluative branch of the metaplan tree representing the agent's comparison between the costs of different plan subclasses. ***Evaluate-subaction***, corollary to ***build-subaction***, models consideration of the cost of the subgoals necessary to the achieving of the current goal.

6.6.2 Higher-Level Evaluative Metaplans

A natural extension to this basic evaluative metaplan scheme would allow for metaplans that captured particular evaluative strategies where there is a further structure to the evaluative process that can provide additional predictive power about the agent's likely follow-on steps. An important example of such structure is when an agent compares two different plans point for point, leading to a very structured pattern of queries. Another case would be an agent trying to select the optimal plan from all the possibilities, which suggests that the queries will cover all possible plans for the goal before moving to a new topic. In this section, we sketch out metaplans for those two cases of more highly structured evaluative planning, showing how they could be added to the existing metaplan model.

A ***compare-plans*** metaplan would model a problem-solving strategy where the agent queries evaluative features of a particular set of plans in a way organized by their features, rather than by their subclass structure. For example, a student using ***compare-plans*** across **earn-credit** plans for a number of different courses (perhaps the answers to an earlier feasibility query about which courses had space available) might ask first about the meeting time of each course, evaluating convenience of access, than about the identity of the professor, and then whether the course has a final exam. Each topic would be covered for all the courses in the set before moving on to a new topic. In fact, it is probably more accurate to say that the queries in ***compare-plans*** are organized around issues that impact the agent, rather than directly around individual plan evaluation features, since a comparison may sometimes span unlike plans with different features that impact the same issues for the agent. For example, a student with particular trouble in speaking that caused her grading anxiety in considering courses might ask about the weight given to class participation in a small course and about the chance of an oral exam in a large one.

There may also be issues that need to be considered in combination, for instance, where the location and time of each course need to be considered together to determine ease of access. Whether phrased in terms of issues or features, it is clear that modeling the structure of a developed comparison would require additional metaplan structure to capture the set of options being compared and the space of possible issues on which the comparison is being based.

The advantage of a ***compare-plans*** metaplan would be that it would supply new links in the tree showing that a closer relationship exists between the plan nodes being compared than that traced in the usual plan tree structure. For example, a consumer comparing two cars might ask a sequence of questions that jump back and forth between the two different variable instantiations, asking the price, the fuel economy, power, and so forth for each of the two. In the normal ***build-plan*** scheme, this would require jumping back and forth between two branches of the tree separated by the different instantiations for the **?new-car** variable. That jumping would be especially difficult if matching the particular queries required building down into subplans and actions below the plan class in which the **?new-car** variable first appears. A ***compare-plans*** metaplan, however, would explicitly encode the comparison that the agent is pursuing. The heuristic component would be arranged so that a jump from one instantiation to another (or to an alternative plan class) would cause the setting up of a comparison metaplan between the two contexts, and any continued alternation then would follow that more direct link between the two contexts.

Another example besides ***compare-plans*** that introduces further structure on a higher level than that of the basic ***evaluate-plan*** is ***choose-optimal***, a metaplan that captures the strategy of an agent who intends to evaluate each of the possible plans for a goal in order to pick the best option. This metaplan when triggered sets up the expectation that each of the options will be explored to at least some extent before a choice is settled on, a strategy that is the opposite of that assumed in the typical ***build-plan*** feasibility case, which settles on the first feasible plan found.

Extending the metaplan model with these higher-level evaluative metaplans would add considerable additional representational power, though it would naturally also introduce additional ambiguity in tracking the agent's metaplanning strategy since the query sequence alone often provides insufficient data to deduce a precise model of that strategy.

6.7 Informing Metaplans

The informing metaplans are different from those discussed so far in that the agent's direct purpose in pursuing these metaplans is to affect the expert's model of the planning space that the agent is exploring or of the priorities that the agent attaches to particular goals or methods within that space. Thus, while the expert may infer from an ***ask-pred-value*** action that the agent is exploring that portion of the domain plan tree that matched the given query, that inference is a byproduct of the agent's metaplan action, rather than its explicit purpose, while with informing metaplans, that becomes the explicit goal.

One theme of the entire metaplan modeling approach, of course, is that

agents are aware of the deductions that the expert will draw from their behavior, even when the explicit purpose of their behavior is plan building, and part of the productivity of the expert advising relationship comes from the agent being able to depend on the expert making exactly those expected inferences. The agent depends upon the expert making reasonable deductions from the agent's explicit behavior about the agent's plans and goals since advice-giving behavior is so heavily dependent on this context that the value of the answers would be strongly compromised if the expert were not working from a model of the agent's goals. Thus conveying enough information so that the expert will have an appropriate model becomes an important part of the agent's own goals, in order to enable cooperative responses.

The agent is usually assisted in this task of ensuring that the expert is able to track her goals by having a plan library that is shared (at least largely) between agent and expert, and by the mutual understanding of the discourse structure of problem-solving dialogue, which allows the agent to predict what the expert will be able to deduce on her own directly from the agent's queries. Based on their own model of this shared data, agents only need to communicate directly about their goals when they feel that the expert will not be likely to be able to deduce the proper context on her own, but they must in those cases add direct statements to communicate the correct context, and that is the role of the inform metaplans.

The agent is constrained by Grice's twin maxims of quantity [21] to offer the expert enough information about context but not too much. The agent must provide enough context so that the expert, combining that explicit information with established expectations in the domain, will be able to build a model that is adequately constrained to resolve any ambiguities that would significantly affect the responses to be made to upcoming queries. For example, in the hardware store query

I want to extend a wrench handle. Do you have any 3/4" pipe?

the context-setting initial statement resolves many ambiguities about the actual query, such as what kind of pipe would be suitable, the approximate length required, and such. In a slightly different way, the toy store shopper who explains that her son wants either a bike or a game, while providing little extra data about the characteristics of the individual goals, does provide a context that makes sense of a conversation that will connect two goals that have no other obvious relationship. That in itself makes the expert's modeling task easier by avoiding wasted energy searching for some inherent connection, and thus does add relevant data.

On the other hand, the agent would also be transgressing the maxim of quantity if she were to provide information about her plans and goals that did not have plausible bearing on the responses the expert would make. For example, if the hardware store shopper were to start explaining why she had decided to redo the bathroom before the new tenants arrived, or if the toy shopper started explaining that making this child happy was especially important because her older sister tended to give her a rough time, they would be adding information that had no relevance to the expert's model building because it is quite unlikely to affect the answers the expert would give to the following questions. Such extra information might be justified in a friendship context by the goal of mutually learning more about the other person, but it is intrusive in the expert advising context exactly because it conflicts with the maxim of quantity.

In this connection, it is interesting to consider the efforts to which agents must go when they wish to prevent the expert from inferring this data about their intended goals, either because of embarrassment or a desire to mislead, where the purpose must remain hidden. Gathering the necessary data without allowing the expert to infer the actual plan requires either that the significant queries be buried among a large number of irrelevant queries, whose purpose is simply to confuse the expert's inferential processing, or that an actual alternative plan be constructed that does not need to be kept secret but that does overlap with the actual plan in terms of requiring the key data. Both strategies sacrifice the benefits that usually result from the expert's cooperative behavior based on her inferred understanding of the agent's goals and plans; the former also warns the expert that the agent probably has some hidden goal, since the expert will soon see that there is no plausible overt goal that could motivate such a sequence of unconnected questions except that of distracting the expert from the meaningful subsequence.

Nevertheless, while agents are aware of the expert's inferential processing that is attempting to deduce facts about their goals from the sequence of queries, that process is usually relegated to the background. The unusual thing about the metaplans in this section is that their explicit purpose is to inform the expert about the agent's goals and constraints. These are the plans used when the agent decides that additional information beyond that which the expert can reasonably infer is likely to be helpful to the expert, and so chooses to provide that additional context directly. These metaplans therefore make explicit the communication of goals from agent to expert that usually happens in the background of the advising process and by inference.

Because these plans make that informing process explicit, we can note two unusual features of these informing metaplans when compared to the other classes covered so far, the first being that they give the agent more control over the expert's model of their goals. While the inferential process can only infer average degrees of positive interest in particular goals from the exploration of plans that would achieve those goals, an explicit inform metaplan can communicate stronger positives, telling the expert that the agent has committed to a particular path or a particular choice of variable instantiation to the exclusion of other alternatives, and also can communicate negatives, ruling out particular subplan options or particular sets of variable fillers, unlike the inferential process, whose strongest negative results from a lack of any demonstrated interest in the particular option.

Second, these inform metaplans are also unique in not causing any structure to be added to the expert's model, because in themselves they only communicate the agent's positive or negative constraints on certain plans, rather than introducing new options. It is true that the expert may only learn on the basis of such an inform metaplan that a particular option was being considered, but it is clearer to understand that as two separate events, an inference from the inform that the given plan is being considered, one that is just like the similar inferences from query actions, followed by a inform metaplan that sets the heuristic measure of the agent's willingness or interest in that plan to either a high or low value.

These informing plans are similar in purpose to the cooperative behavior described by Joshi, Webber, and Weischedel [30]. There they pointed out the responsibility of cooperative speakers to block incorrect assumptions on the part of

their hearers that would otherwise lead their hearers to an incorrect model of the speaker's intent. In order for the conversation to proceed felicitously, the speaker needed to be sure not only that what she was saying was correct, but also that the hearer's conclusions based on it would be correct. These inform metaplans similarly are attempts on the agent's part to ensure that the model the expert will derive from the sequence of queries will match the agent's actual intentions.

It is also interesting to observe that the tradeoff between informing metaplans and the normal plan-building and query ones is affected substantially by the agent's perception of the knowledge and cooperativeness of the expert. Users of traditional computer database systems, for example, knowing that the expert in that case is able to answer queries but is not maintaining any plan context model, would have no reason to attempt informing metaplans, just as there is in that context no reason for them to try to ensure that their query sequence is suggestive of their goals and not misleading. Agents consulting with cooperative human experts, on the other hand, know that they can gain through informing actions, and may even use informing as a way of asking the expert to actually build the plan, rather than just to supply the data for the agent to use in that process. As NL systems become more sophisticated about maintaining and using a model of the agent's intentions, agents may come to realize the usefulness of also informing that kind of expert about their goals.

The inform metaplans can be divided into two classes, ***inform-goal*** and ***inform-constraint***. Their effect is to replace the former representation of the current context as a node in the metaplan tree (or more than one if the previous match was ambiguous) with a representation for the new context implied by the inform actions. This could be implemented in the model at least for straightforward examples by having ***inform-goal*** be a child of ***build-plan*** and ***inform-constraint*** of ***constrain-var*** so that the inform nodes would be associated with the plan-building nodes describing the appropriate plan class or constraint. The inform metaplans are the only nodes that will match with the partial interpretations of inform utterances, both because their form is different from the normal query form and because their body contains plan class or constraint descriptions, so that an agent's explicit goal statement, for example, would cause a search through the tree for a ***build-plan*** node with an ***inform-goal*** node attached whose plan class description would match with the plan class specified in the utterance. A match to that ***inform-goal*** node would then automatically have the effect of resetting the current context to be that described in the goal-informing statement.

6.7.1 Inform-Goal

The ***inform-goal*** metaplan is used when the agent wants to steer the expert's modeling of the agent's planning process, providing explicit indication of the goal that the agent is attempting to achieve, in cases where the agent judges that the expert would not otherwise be able to deduce it. In the naval domain, if the damaged ship was out of commission due to a broken radar, the agent might specify,

I bet Wilson has the spare parts. How far away is it?

While that query includes an ***instantiate-var*** to Wilson followed by an ***ask-pred-value*** for the distance between it and the damaged ship, it also goes out of

its way to make clear what plan the agent in considering, namely, the use of Wilson to supply the spare parts as part of repairing the damaged vessel, rather than as a replacement ship itself.

The additional information of an *inform-goal* is attractive to the agent whenever it seems that the expert would otherwise have difficulty for whatever reason supplying the plan context correctly. In the example above, the reason for the *inform-goal* comes from the ambiguity between use of the Wilson as itself the replacement ship (since it fits all the requirements for one) or its use only as a source of spare parts for repairing the damaged Sterett. The agent is aware of the expert's plan model and therefore of the danger that the expert will choose the closest domain plan into which the query about the Wilson could be fit, and thus misidentify the plan.

Thus, examples of *inform-goal* are especially likely in cases where the plan is unusual or ambiguous. However, even in cases where the expert is likely to be able to identify the agent's plan correctly from the query itself, if that possibility would be only one among a number that the expert would have to carry along, then the reduction of ambiguity and the associated easing of the expert's modeling task may itself be worth the effort of a goal informing.

6.7.2 Inform-Constraint

The other informing metaplan is *inform-constraint*, where the agent informs the expert explicitly of a constraint she wishes to apply to the possible fillers of a particular open variable slot. Just as *inform-goal* allows the agent to explicitly specify particular plan subclasses, *inform-constraint* does the same for restrictions on variable values. For example, a naval agent who had already used *inform-goal* to say

I want to replace the Sterett.

could continue with *inform-constraint* by saying

I want to use either Fox or Perry.

Just as with *inform-goal*, *inform-constraint* allows the agent to make explicit statements about exactly the sort of plan modeling issues that the expert is usually left to deduce from the queries and the context. In this explicit form, naturally, the agent has more control over the goal structure communicated, and can communicate quite complex constraints. However, since the deductive style usually provides adequate context, problem-solving agents only resort to inform actions where unusual circumstances of ambiguity or complexity suggest that the extra information will significantly improve the expert's ability to respond helpfully.

6.8 The Combined Metaplan and Domain-Plan Model

In this last section, we describe briefly how the metaplans we have been considering can be used to build a tree that represents the state of the agent's planning process at the metaplanning level, in much the same way that the domain plan tree of domain plan classes looked at in the previous chapter models the agent's action plan. In fact, we will see that the metaplan tree ends up including enough information about the domain plan tree that it can itself serve both purposes.

First, then, the subplan and subaction relationships among the metaplans guide the ways that the metaplan tree can expand. For example, while many plans can be direct children of ***build-plan***, a branch containing a ***limit-cardinality*** query has to end with an ***ask-fillers***. This models there being many different approaches to building a plan in general, but also captures the internal coherence among elements of a line of inquiry once initiated. Relevance of the particular metaplan to the metaplanning situation is captured, as expected, in the constraints of the metaplan, which require, for example, that an ***ask-fillers*** branch can only be initiated beneath a ***build-plan*** for a plan that contains open variables not bound at that point in the tree by any earlier ***instantiate-var***.

For our purposes, the tree of plan-building, evaluative, and query metaplans includes all the necessary data that would be found in a separate model of the domain plan tree, and thus subsumes it. As we have seen, the central plan-building metaplans include arguments for the domain plan being considered and for their arguments, including the values being considered for any open variables in the domain plans. Thus the complete domain plan tree can be found by following these links from the metaplan tree nodes. There is no reason for our purposes here to maintain an independent model of the domain tree.

One reason why this simplified model is possible is because domain modeling of expert advising interactions can ignore the state of execution of the domain plan. During the consultation, we assume that the agent is not taking any actions that change the problem state, so that the problem and the possible actions to resolve it can be viewed as a fixed context for the plan-building and queries. Thus, we get away here with a single world model to describe the situation, against which the trees of alternative plans can be built. This assumption of a fixed world during the consultation does not mean that the domain plans in a metaplan model like the one presented here cannot represent change. The addition of time indices to the formulas would make it possible for them to represent future changes caused by the plans under consideration and even to deal with the problems of interacting plans. Building a plan context to handle questions like

Is CIS 214 offered in the spring?

for example, requires that the system be able to conclude that it is feasible to take CIS 214 in the spring given that one is enrolled in CIS 213, its prerequisite, in the current fall semester. When the times of future actions are not specified fully, there may be ambiguity about the answer to be given depending on how the times are instantiated. Thus the answer to the query

Will Frederick be in Honolulu on Monday?

certainly depends on whether other actions involving that ship are undertaken in the meantime. All of these general issues in temporal planning and interacting plans also apply to the domain planning underlying the metaplan problem-solving model even though the specific issue of changes in the world from current action can be avoided.

The metaplan tree thus is able to serve as a model of the pragmatic context both on the domain planning level, where the current plan subclass and variable instantiations capture the class of plans that the agent is considering, and on the problem-solving level, where the metaplan tree models both the subclass and variable instantiation refinement steps that the agent has used to arrive at the current plan and the precondition and set filler queries the agent has asked to gain information about the feasibility or relative cost of the various plan options. The ability of this model to capture and predict the agent's problem-solving process is the heart of Pragma's approach to applying pragmatic knowledge to resolving ill-formedness.

CHAPTER 7

LINKING TO PRAGMATIC CONTEXT TO RESOLVE ILL-FORMEDNESS

In this chapter, we consider how to use the pragmatic context model based on metaplans described in the previous chapter for resolving particular classes of ill-formedness. In broad terms, the approach is to use the pragmatic model to generate a space of predictions for the possible next plan-building moves on the agent's part and for possible queries stemming from those moves, and then to search in that space for nodes that can be linked to the actual ill-formed query, where "linking" is used for the process of identifying possible matches with the partial interpretation representing the understood portion of the ill-formed query. The plan-building, set, query, and informing metaplans presented in the previous chapter, along with the library of domain-level plans, are used to generate the prediction space of possible plan-building moves and queries. This chapter presents the heuristic principles and methods that can be used to rank the plan-building and query nodes, adding a probability metric to the search space to help direct the search toward the most likely links.

This heuristically-guided search for links takes place in the space of possible problem-solving moves outlined by the metaplan model operating on the domain plans relevant to the known top-level plan context. The advantage of that model is that it does map out the plausible follow-on queries that can result from the agent's consideration of related topics in the plan building process, limiting the set of possible queries that needs to be searched to those that are logically related to the former context and imposing a structure on that space that makes it much easier to search. Thus the theory of the metaplan model provides the core of knowledge needed for the linking approach.

However, the space of possible continuations in expert advising dialogue is rich enough that the hard constraints provided by the metaplanning model in terms of what queries can be asked still leave a space of possibilities that is too large to search exhaustively. The structure of domain plan classes and actions outlined by the plan-building core is already large, each of those ***build-plan*** nodes can be expanded in turn by any of the applicable query generating plans, and the ***constrain-var*** plans can even introduce arbitrary further constraints, making it impossible to fully explore the space of metaplan nodes accessible from any given node.

Because the metaplan model can generate a very large number of possible plan-building moves and queries from any particular context, practical use of this linking approach requires well-developed heuristic methods for directing the search and for ranking the different links identified. Even if the input being tracked was well-formed, heuristic guidance would be necessary, since there are frequently multiple metaplan derivations from a given context even for the same well-formed query, derivations that assume different underlying plan-building moves on the agent's part, even though the resulting query turns out to be the same. For instance, in the example

from the previous chapter where the Sterett was damaged, the query

Where is the Frederick?

could be generated either by consideration of a plan to replace the Sterett with the Frederick or by a plan to use the Frederick to deliver spare parts so that the Sterett could be repaired on station.

This need for heuristic control is even more apparent in working with queries that are ill-formed, so that only a partial specification of the parse for the query is available, a partial specification that, naturally, may seem to link to many query nodes that the fully interpreted query would not have linked to. Indeed, as discussed in Section 2.1, relaxing the rules of well-formedness in any context can make the search space much larger and greatly increase the number of candidate solutions, to the point that if arbitrary relaxations were allowed, a complete search could identify any particular sentence as an ill-formed way of expressing any possible thought. Even though the single-word error assumption prevents things from coming quite to that extreme, still this process of linking to the predictions of the pragmatic model is particularly dependent on heuristics when working with queries that are ill-formed.

The feasibility of linking for ill-formedness resolution is due to the full exploitation in the search of the heuristic expectations generated both by the pragmatic context and by the partial interpretation of the ill-formed query. Therefore, there are both heuristics discussed in Section 7.2 based on the pragmatic context tree itself, predicting the likely moves within it based on the previous context and independent of the actual current query, and also heuristics discussed in Section 7.3 based on the partially understood query which further constrain and direct the search toward areas of the tree where links might be found. This combination of heuristic methods provides a degree of robustness not attainable with either knowledge source independently.

Throughout this chapter, we will see that the heuristics that are used in identifying links between the partial interpretation of an ill-formed query and the pragmatic context grow directly out of those that are used in the normal query-to-query updating of the plan context tree. The heuristics of tree distance and context and those based on the query form and content operate in the same way, in fact, whether the input query is well-formed or ill-formed, with the only exception being that the partial interpretation of an ill-formed input often is not able to supply the data necessary for the operation of some of the query-centered heuristic constraints. Thus, the bulk of this chapter presents the heuristics as they operate in tracking well-formed queries, and Section 7.4 later discusses the necessary modifications in the heuristics when linking to ill-formed queries.

7.1 A Framework for Heuristics in the Metaplan Tree

Although various kinds of heuristics are used in the Pragma system to rank the likelihood of the possible plan-building moves and queries and to direct the search for a link to the actual ill-formed query, a single representational scheme was chosen, designed to be powerful enough to serve for all of them, namely a combination of scores attached to each metaplan tree node with a form of condition/action heuristic rules or "h-rules". The condition tests of the h-rules can examine the existing score of

the node, the neighboring metaplan tree structure, and the instantiation of metaplan variables to elements of the domain plans or domain model constants, while the action parts of the rules can either reset the score for the current node completely or else alter it by adding or subtracting a given amount. The score for a node is determined when that node is reached in the exploration of the metaplan tree by testing each of the h-rules against the conditions at that node and executing the action parts of those rules that succeed, resulting in the assigning of a score to the node.

The function of the heuristic scores assigned to the metaplan tree nodes by the h-rules is to impose a probability metric on the search space of possible plan-building moves and queries, so that the search process can focus first on those portions of the tree that represent the most likely moves for the agent in the given context. The search process is driven from a sorted agenda that chooses for expansion at each step the unexplored node in the metaplan tree with the highest heuristic score, so that the most likely expansions of the metaplan tree as judged by these heuristics are tested for possible links first. In judging that likelihood, the h-rules make use of information from various sources, as the rest of this chapter will show, including the shape of the metaplan tree, the particular configurations of nodes surrounding the node being scored, and clues to possible links found by comparing the node context with the partial interpretation of the current query.

While examples of the h-rules will be presented in this chapter as they are implemented in Pragma, it is important to stress the difference between the implemented form of the h-rules and the heuristic principles they embody. The significant elements in the heuristics presented in the rest of this chapter are the source and kind of information considered relevant and the direction and degree of influence that the given fact is considered to have relative to the other heuristic influences. The elements of the actual encodings for the h-rules that will turn up in the examples like the specific scores given to particular rules are only meant as a general indication of the strength and combining power of the given heuristic influence. While the details of the combining scheme for heuristic scores will have an important effect on the performance of any particular implementation of this linking approach, those details will always be dependent on particular features of the implementation. The sources and directions of heuristic guidance that are useful in this situation, on the other hand, are factors that will be true independent of implementation.

7.2 Probable Query Prediction: Heuristics from the Context Itself

There are three classes of heuristics that distinguish between the likelihood of various next steps based on the current context alone, apart from the actual query: first, a set of general heuristics dependent on distance in the tree as a measure of relevance and coherence and on encoding in the heuristics typical problem-solving strategies to help direct the search of the plan tree, second, a set of more specific heuristics predicting the agent's choices between different metaplans based on a model of her knowledge about the current state of the world, and third, heuristics that select between different instances of the same metaplan based on features of the domain plans involved. All of these heuristics, of course, are built on top of the metaplans themselves, which define the basic shape of the space of possible plan-building actions and queries within which the agent's planning behavior is interpreted, but while the

metaplans are intended to provide a complete description of the space of possible problem-solving actions, the focus in this chapter is on a heuristic structure of probable expectations that can restrict that space of possible problem-solving behaviors.

7.2.1 Heuristic Use of Tree Shape and Problem-Solving Patterns

The shape of the metaplan tree is used in a number of ways as a model of the agent's problem-solving process that can direct the plan-tracking search for the most likely node in the tree corresponding to a particular problem-solving action. The fundamental heuristic here is to interpret the metaplan tree as an analogue of the agent's problem-solving space, so that nearby nodes in the tree refer to issues closely related to the current context in the agent's process. This use of tree distance as a heuristic would suggest a breadth-first search in the metaplan tree working outward from the known previous context. That simple model is then refined by modifying the breadth-first search so as to better predict the typical problem-solving behavior of human agents, and especially the depth-first exploration of subproblems that is part of typical human problem-solving coherence. Finally, we point out how the model can account for other, non-typical problem-solving patterns and adjust its search strategy to account for them.

7.2.1.1 Tree Distance as a Model of Coherence

The main heuristics used to predict relevant continuation queries based on the plan context as encoded in the tree derive from the shape of the tree around the current context by implementing a modified form of breadth-first search. Such heuristics are implemented in Pragma by default rules like the following **default-downward-subplan** rule that initializes the score for a new ***build-subplan*** node by taking the score of its parent ***build-plan*** and reducing it by a small, fixed amount, as shown in Figure 7.1.

```
(def-h-rule default-downward-subplan (?node)
  conditions ((subplan-of ?parent-node ?node)
              (node-score ?parent-node ?parent-score))
  actions ((set-score ?node ?parent-score)
           (dec-score ?node 5)))
```

Figure 7.1: Default-Downward-Subplan H-Rule

Here the first condition identifies the parent-node, the second, when unified against the database of search data, retrieves the parent's score into the variable **?parent-score** (unless the parent has no score assigned yet, in which case this rule does not fire), and the actions set the child node's score to that of the parent less 5. There are default rules similar to that given above to cover each of the kinds of links in the metaplan tree in both downward and upward directions. The effect of these default rules is to score more highly those portions of the tree nearest to the context where the search is begun.

These heuristics are based on a double approximation, first that distance within the plan tree can be used in some form as a measure of logical relatedness, and second that logical relatedness can be used to predict likely follow-on queries. Thus

one searches the nearby portions of the plan tree first, believing that they represent metaplan processes closely connected logically to the current context, and thus more likely than more remote nodes to be able to be matched with the agent's follow-on queries.

The first step here, the use of distance in the metaplan tree as a measure for logical relatedness, is implicit in the spreading activation style of the search process for links in Pragma, which organizes the search largely as a breadth-first search working out in the tree from the current context. Note that this sense of distance in the plan tree is not one that can easily be logically formalized, since the number of layers in a classification hierarchy depends on the fineness of the distinctions made, so that the number of steps of subplanning separating, say, **restore-battle-group-readiness** and **replace-ship** depends on the amount of elaboration that has occurred over the levels in between, and thus whether possible intermediate plans like **restore-group-readiness-using-new-equipment** or **restore-group-readiness-single-replacement** will exist as their own layers in the metaplan tree. Thus, the semantic distance covered by each layer of the classification hierarchy of plans is somewhat arbitrary. It is a reasonable heuristic measure nonetheless, since all that is necessary for these heuristics to work is for there not to be large differences between various branches of the tree in terms of how densely subdivided they are. That is, only the relative, rather than the absolute, measure of depth is used heuristically by the system, and that measure will be useful as long as the different areas in the plan tree are treated consistently.

Another use of the shape of the metaplan tree, in addition to selecting which metaplan nodes are explored first, is to provide a heuristic cutoff beyond which the search can be terminated. Such a cutoff is necessary, since the size of the metaplan tree is in principal unlimited, and even in practice, that tree will be very large, so that a complete search across it would be prohibitively expensive. Thus any practical system has to rely on some heuristic judgement of how much of the tree should be searched to find candidate matches for a particular utterance. The fact that the tree structure can be taken as an analogue to some degree of the problem structure in the agent's formulation means that it is quite unlikely that the correct match to a closely related query will be found far away in the tree, so that the assumption that the agent tends to explore logically related topics justifies the use of a tree distance heuristic to restrict the search.

Note that the gradually decaying scores assigned by these default tree distance heuristics are only one factor in assigning scores to newly explored nodes, so that it is quite possible that a more highly-ranked match can be found farther from the previous context than a lower-ranked one, and heuristics based on other factors can also help to cut off a particular search path more quickly than the default tree distance alone would have done. However, these default h-rules provide an underlying breadth-first character to the search that focuses it in areas closely related to the previous context.

7.2.1.2 The Typical Problem-Solving Pattern

While distance as measured in the metaplan tree can model a general sense of logical relatedness, there is further structure to the agent's problem-solving process that the heuristic component must also try to capture. There are patterns of exploration within the problem-solving tree, and by recognizing those patterns, the system can predict more accurately the agent's likely follow-on queries. The pattern of exploration tends to vary with the class of metaplan involved, with the normal plan-building metaplans showing a typically depth-first exploration pattern, as discussed in this section, while the evaluative metaplans evoke the more comparative, breadth-first one covered in the following section.

The typical exploration pattern for the aspect of problem-solving behavior captured in the plan-building metaplans appears to follow a depth-first approach in which one subproblem is worked through to completion before the agent begins on another subproblem. In the plan-building context, this means that all the plan feasibility queries relating to one domain plan class and the actions it entails are likely to be asked before any relating to a different plan class. This depth-first approach seems to apply both at the ***build-subplan*** and ***instantiate-var*** nodes that are logically OR nodes and at the ***build-subaction*** AND nodes. As for the former, given that the agent is searching for a feasible solution, it makes sense that she would not abandon any of her tentative choices to refine or partially instantiate the proposed plan unless and until those choices had been shown to be infeasible. That much implies that alternate plan choices will not usually be explored until the previous option has failed to pan out. The depth-first pattern also seems to apply even at AND nodes, where each branch will eventually need to be explored, and here the justification is probably more one of efficiency in terms of focus of attention, that a depth-first pattern requires fewer context-switches between different parts of the problem.

Remember that there are two mutually-supporting arguments for why human agents tend to use a coherent and predictable problem-solving strategy, in this case, following a depth-first problem organization. On the one hand, as just mentioned, that pattern of exploration may be the most efficient from the agent's point of view. On the other hand, regardless of what is easy from the agent's point of view, the agent is driven to using a coherent strategy exactly because that is easier for the expert to track. The tracking effort and number of context switches required of the expert is as important a factor in determining a preferred problem-solving strategy as that required of the agent. The expert's ability to track the agent's goals is important both for the expert, in terms of her understanding about the situation, and for the agent, since the expert's ability to be helpful is strongly dependent on the extent to which the expert understands the agent's goals. As discussed in Section 2.2, it is that knowledge of tracked pragmatic context that makes many sorts of cooperative responses possible, as well as strengthening the expert's hand at interpreting anaphora or ellipsis or other ambiguous or ill-formed input.

This predictability and coherence in human problem-solving strategy is markedly different from the strategy one might imagine for a computer problem-solving program, where the ordering constraints might be very different. An automated reasoner could be programmed to ask at each stage the query that would

contribute the fact most useful at that point to the plan-building process, regardless of its degree of connection with the previous query. Such a strategy might well seem optimal from the computer's point of view. However, that random strategy makes much more difficult exactly that tracking by the expert of the user's goals that we are modeling here. Only a database system that made no effort to track pragmatic context would be comfortable as the expert serving such an agent. Experience with expert diagnostic systems has born out this demand on the part of human beings who are trying to track the progress of the agent's plan building; systems like MYCIN [49] that were first programmed to ask their queries in this random style where the next query was the one most useful to the computer's diagnostic process proved so frustrating to their human informants, who were trying to impose a logical problem-solving organization on the sequence of queries, that their control structures had to be modified to enforce that sort of coherence. A theme of this current research, of course, is that even computer systems answering the questions of problem-solving agents depend on the agent's adopting a coherent exploration strategy in the metaplan tree in order to track the agent's goals and respond cooperatively. And that pattern of coherence must not only be abstractly logical, but also conventionally understood.

The tree-search heuristics in Pragma model this depth-first exploration pattern for plan feasibility by giving a high weight to subplans and subactions beneath the current node, a higher one than for moves to consider siblings to the current plan, and those in turn a higher rating than moves which lead through ancestors further back on the tree than the parents of the current plan. These weights are assigned by a family of h-rules including the **default-downward-subplan** h-rule shown earlier that alter the weights assigned to nodes away from a strict breadth-first strategy in favor of one that captures this sort of depth-first coherence. Each of the rules assesses a cost for an arc by decrementing the score derived from the previously-explored node by a small fixed amount before assigning it to the new node. Embodying the heuristic that the agent is most likely to pursue a subplan of the current plan, the **default-downward-subplan** rule uses the smallest decrement of any of these default tree distance rules, a factor of 5 in the 100 point scoring scheme, since it merely represents a further specification in the agent's domain plan of the class of plans being considered. The **default-downward-subaction** h-rule shown in Figure 7.2, which models a change in the agent's focus from a class of plans for a particular goal to plans for one of the constituent actions shared by the plans in that class, represents a more significant step and carries a decrement of 10.

```
(def-h-rule default-downward-subaction (?node)
  conditions ((subaction-of ?parent-node ?node)
              (node-score ?parent-node ?parent-score))
  actions ((inc-score ?node ?parent-score)
           (dec-score ?node 10)))
```

Figure 7.2: Default-Downward-Subaction H-Rule

While the low costs of subplan and subaction arcs represent the greater likelihood of the agent continuing to consider the plan class currently in focus, relatively higher costs are charged to exploration paths that move up from the current node to consider alternative plan classes to the one currently in focus or their subplans or actions. The **default-upward-expansion** rule itself assesses a decrement of 5, which is in addition to the inherently longer search path in the metaplan tree required to

reach such nodes. The net result of these varying costs is to direct the search of the metaplan tree following the typical depth-first pattern so as to explore first those areas that the agent is most likely to consider next, given the current context.

This kind of coherence in human problem-solving activity has naturally been observed and made use of in previous plan-tracking systems, which make the same double assumptions about the plan tree as a model for logical relatedness and relatedness as a predictor for follow-on queries. For example, Carberry's TRACK system [10] includes rules that embody exactly these heuristics for a depth-first problem-solving approach, only moving on to other subproblems or more distantly related possibilities when the current one has been sufficiently explored. A key advantage of using domain plan trees as a model has always been their ability to predict problem-solving behavior, an ability that stems directly from their exploitation of this heuristic. The difference in this current approach, however, is that here the measurements of coherence are defined on the metaplan tree of problem-solving plans, rather than directly on the domain plan tree, and this significantly improves their predictive power, since the metaplan tree reflects levels of logical organization in the problem-solving behavior that are not modeled in the domain-plan tree. That advantage becomes the key in the next section, where we show how particular problem-solving patterns other than the default depth-first exploration can be handled in the metaplan context because the metaplan tree can capture the agent's alternative problem-solving organization, introducing structure that shows the close logical links between queries that would otherwise appear widely spread in the metaplan tree.

7.2.1.3 Representing Alternate Problem-Solving Patterns in the Tree

The previous section on the typical problem-solving strategy of agents using expert advising systems presented heuristic rules that implement the strategy found in the set of metaplans that describe the plan-building activity of an agent whose goal is merely to find a feasible plan. The standard set of heuristics based on the ***build-plan*** metaplan with its ***build-subplan*** and ***ask-pred-value*** children represents a problem-solving strategy whose aim is to identify any complete plan for the given goal. That is why, within a single goal, further exploration of a partially-explored branch is heuristically preferred, and why, except when a precondition query has received a negative answer, queries about other alternative subplans are heuristically discouraged.

While that strategy is the default, there are other strategies for problem-solving that take a different approach, where the agent intentionally explores multiple possible paths in order to evaluate their costs and thus be able to choose the cheapest or most beneficial option. Those evaluative strategies sometimes select a single criterion to measure from each plan possibility, like a naval planner testing each option, say, shipping spare parts vs. replacing the vessel, to find the one that will get the damaged vessel's task force back into operation with the least possible delay. Other evaluative strategies call for comparing the different plans point by point on multiple features, as if constructing a table comparing the options. As described in Section 6.6, these evaluative strategies are encoded first in the ***evaluate-plan*** metaplans, which function like ***build-plan*** to create plan tree structure and to match user query input, but which are not heuristically favored except when particular circumstances in the

discourse cause the heuristics to activate them, and that section also outlined metaplans to capture higher levels of structure in evaluative strategies like comparing plans. This section discusses the exploration patterns that go with these evaluative metaplans and the heuristic means used to activate them in the cases where they apply.

Given the definition of the *evaluate-plan* set of metaplans, the first job of heuristics is to activate these unusual strategies in and only in the appropriate circumstances. The alternative strategy metaplans are usually not explored because the default heuristics assign them much lower scores than the standard continuation metaplans. For example, the default heuristic rule that activates ***evaluate-plan*** from a ***build-plan*** base in Figure 7.3 assigns it an initial weight of only 20% of that of the ***build-plan*** node.

```
(def-h-rule default-eval-plan-from-build-plan (?node)
  conditions ((node-ps-plan-name ?node evaluate-plan)
              (node-parent node ?parent)
              (node-ps-plan-name? parent build-plan)
              (node-score ?parent ?parent-score))
  actions ((set-score ?node (* ?parent-score 0.2))))
```

Figure 7.3: Evaluate-Plan-from-Build-Plan H-Rule

In spite of the low initial score given this metaplan, it is usually easy for the system to identify ***evaluate-plan*** queries because the assertion involved in the query is not one of the domain plan's preconditions, but instead involves one of the set of evaluative functions that is stored with each domain plan. For example, while the readiness of the replacement ship is coded as a precondition of the **replace-ship** plan, it's rate of fuel consumption, which affects the cost it will incur in sailing to join the task group of the damaged ship, is not listed as a precondition, but is still a function of the plan that becomes relevant when the user wants to compare the cost of this plan with that of other feasible plan options. Thus, the initial activation of the ***evaluate-plan*** metaplans usually happens automatically in the search for a query node match, since only the query nodes attached to ***evaluate-plan*** nodes can match with these evaluative predicates.

While the primary way the model can recognize evaluative contexts is through queries about evaluative predicates, it is also possible for particular patterns of query behavior themselves to alert the heuristic component to the likelihood of evaluative queries. For example, a series of queries about preconditions within one subplan that all succeed suggests that the user has discovered a viable approach, and that she will next look for plans for other required actions, rather than at other alternative subplans for this apparently successful goal. If the user continues to ask about other possible plans after seemingly establishing the feasibility of one, the system can recognize that the agent is not following the normal plan feasibility pattern, and one of the consequences of that is to increase the likelihood of evaluative queries about those two subplans. Thus if a user who has explored the use of one replacement vessel with apparent success (in that all queried preconditions succeeded) goes on to explore another, a special heuristic can detect the exploration of a second sibling to a successful branch with the effect of increasing the ranking of evaluative metaplans for those plan branches.

Once the model has tracked the presence of an unusual problem-solving strategy like *evaluate-plan*, the heuristics can then take account of that in the rankings for further continuation queries. Thus, once the current node is an instance of *evaluate-plan*, the following two heuristics then work to give high scores to *evaluate-plan* nodes for sibling plans.

The first such evaluative pattern heuristic gives a boost to *evaluate-plan* nodes on alternate subplans to the current plan, as shown in Figure 7.4.

```
(def-h-rule eval-plan-sibling-build-plan (?node)
  conditions ((node-score ?node ?node-score)
              (node-ps-plan-name ?node evaluate-plan)
              (node-parent ?node ?parent)
              (node-parent ?sibling ?parent)
              (node-ps-plan-name ?sibling
                                 evaluate-plan)
              (not-equal ?sibling ?node)
              (node-score ?sibling ?sib-score))
  actions ((set-score-to-max ?node
                             ?node-score
                             (- ?sib-score 10))))
```

Figure 7.4: Evaluate-Plan-Sibling H-Rule

The set-score-to-max operator here has the effect of increasing the score for the node assigned so far by the default tree exploration rules, if lower, to 10 less than the score of the *evaluate-plan* node for the sibling.

The second heuristic of this sort, shown in Figure 7.5, handles cases where the alternative node to receive the boost is an alternate instantiation of a variable from the current plan. In this case, the score of the current node is brought up if necessary to 15 less than that of the equivalent *evaluate-plan* node in the already-explored branch with the other variable instantiation. The effect of these two h-rules is to have the scores for *evaluate-plan* nodes carry over to other *evaluate-plan* nodes that are more closely related to them by the evaluative pattern than would be apparent from the metaplan tree structure itself, modeling the typical use of these evaluative plans in choosing between alternate feasible plans.

The metaplans discussed at the end of Section 6.6 that attempted to capture higher levels of structure in the agent's evaluative problem-solving would also require matching heuristics to model the appropriate exploration strategies. As with *evaluate-plan* itself, the default heuristics working from the *build-plan* tree would initially discourage moves from there to evaluative contexts like *compare-plans*, but once an evaluative context became established through a sequence of queries, the heuristics would encourage further exploration within that context. Thus, for *compare-plans*, the initial move from one alternative to another is tracked by the usual path up through the *build-plan* node and down the alternate branch to the other plan or variable instantiation. While that first move is tracked by the normal plan-building heuristics, is also causes a *compare-plans* metaplan to be set up linking the two. From then on, because of that new comparison node, the heuristics in further searches will favor continuing exploration of the comparison.

```
(def-h-rule eval-plan-alternate-instantiation (?node)
  conditions
   ((node-ps-plan-name ?node evaluate-plan)
    (node-score ?node ?node-score)
    (node-parent ?node ?parent)
    (node-parent ?parent ?grandparent)
    (node-ps-plan-name ?grandparent instantiate-var)
    (node-parent ?grandparent ?greatgrandparent)
    (node-parent ?greatuncle ?greatgrandparent)
    (node-ps-plan-name ?greatuncle instantiate-var)
    (not-equal ?greatuncle ?grandparent)
    (node-parent ?uncle ?greatuncle)
    (node-parent ?cousin ?uncle)
    (node-ps-plan-name ?cousin evaluate-plan)
    (node-score ?cousin ?cousin-score))
  actions ((set-score-to-max ?node
                             ?node-score
                             (- ?cousin-score 15)))
```

Figure 7.5: Eval-Plan-Alternate-Instantiation H-Rule

7.2.2 Predicting Plan Tree Growth from Agent's World Knowledge

While the default heuristics presented in the previous section controlled the search by means of distance in the metaplan tree and by means of particular problem-solving patterns as modeled by the metaplans, the heuristics described in this section are intended to take advantage of situations where the expert is able to predict something about the agent's knowledge and where that prediction changes the default probabilities for the particular metaplans in question. These heuristics can therefore help predict the likelihood of particular metaplanning actions in particular circumstances.

Like the plan tree shape heuristics, these agent's world knowledge heuristics also depend on the metaplan model's predictions of which queries can coherently follow from a given context. While the tree-shape heuristics from the previous section weight the nodes in the tree by distance from the current node, adjusted for problem-solving pattern, these heuristics add a relative ordering to the possible nodes at a particular branch point in the tree based on the expert's understanding of the agent's world knowledge. Like the default tree distance rules, these rules are also coded as condition-action rules whose firing changes the heuristic score of that node. Their left-hand sides can test the usual features of particular metaplanning contexts like which metaplans are active at that particular node or at its neighbors or ancestors and the assertions that are relevant to those metaplans, but the left-hand sides of these rules can also test assertions against a model of the agent's world knowledge, to determine whether the expert does or does not have reason to believe that the agent is aware of the truth or falsity of that particular assertion.

A good model of the agent's world knowledge can add substantially to an expert's ability to track the agent's plan-building process in a fluid and cooperative

manner. Such an individualized model of the agent's world knowledge allows the expert to customize the planning model, enabling more accurate and efficient prediction of the agent's most likely moves. Examples of this can be seen in human interactions when planning assistance is given by a person who knows the agent well enough to be able often to predict exactly which subplans will be considered and which queries asked. A large part of this expertise comes from a well-developed model of the agent's domain knowledge, allowing the expert to predict accurately which possible plans will seem feasible to the agent as opposed to those she will know are not possible, and which preconditions for those possible plans the agent will not already know the status of and thus will be likely to query.

Because of the brevity of the expert advising situation, such complete and individualized models of agent knowledge are not possible. Because the expert is assumed to be providing data to agents in relatively short, unconnected sessions, there is no opportunity to develop over time a detailed picture of a particular agent's domain knowledge. Thus, the heuristics in the following sections are based in general not on carefully acquired facts about a particular agent's knowledge, but instead on heuristic judgments about the likelihood of a typical agent knowing some special class of facts. Of course there are cases where certain facts about the agent's domain knowledge will become evident, even in the limited course of such a consultation, and these heuristic rules do take appropriate advantage of such knowledge when it becomes available.

There are three classes of heuristics that use predictions of the agent's world knowledge to influence the weights assigned to various metaplans, based on the kind of metaplanning involved. One class uses predictions of the agent's knowledge about the assertions that are preconditions for domain plans referenced in the metaplan tree to influence the rankings of the plan-building metaplans applied to them, the ***build-subplan*** and ***build-subaction*** plans. The second uses similar predictions of the agent's knowledge to influence the rankings for query metaplans growing out of those subplan and subaction metaplans. The third uses predictions of the agent's knowledge about the size of the set of possible fillers for an open variable in a plan to control the rankings of the variable instantiating plans and their queries. These three classes are discussed in succeeding sections, after some further introductory sections that consider the different classes of agent world knowledge that the system tries to model for use by these classes of heuristics.

7.2.2.1 Agent World Knowledge vs. Plan Knowledge

In this section, we begin to characterize the agent world knowledge accessed by this class of heuristics by contrasting it with the world knowledge embodied in the plans. While the metaplan model does assume that the agent shares with the expert the same definitions of the possible domain plans and of the preconditions that apply to those plans, it makes no assumptions about the agent's knowledge concerning the actual truth of those preconditions for particular arguments. For example, the model does assume that a registering student knows that getting a B.A. degree requires getting credit for a certain number of courses and that getting credit for a course usually requires enrolling in it and attending the classes, but it does not assume that the student would know what particular courses were being offered this semester, which had space available, or what their meeting times were.

The metaplan model's assumption of the agent's ignorance about the state of the world is what allows it to model that the agent might pursue any plan that the plan definitions themselves would allow for in any world state, and might ask queries about the truth of any of those preconditions or possible variable values. That default assumption of across-the-board ignorance is modified by the heuristics presented in this section so that the exploration of the metaplan tree can be directed toward those subtrees that the agent is more likely to pursue given this newly-added knowledge or set of assumptions on the expert's part about the state of the agent's world knowledge.

Before going on to discuss the agent world knowledge model that is added for these heuristics, it is important to say a bit more about the implicit world knowledge model included in the assumption that the agent has perfect knowledge of the plans and their preconditions. This is a simplification that will have to be relaxed in later work, since there are clearly cases where the agent is unaware of a precondition of a plan, like a student who thinks she is ready to graduate but has forgotten to file an application for the degree. There are also cases where a planning agent using an expert advising system may ask queries that, as formulated here, would be queries about plan preconditions rather than about the facts of the domain, like a student asking how many course credits are required, so that the agent's knowledge of the plans can even become something that must be modeled explicitly in order to explain the query behavior. A model of plans in terms of user beliefs is required, as Pollack [41] has pointed out, to be able to handle cases where the agent's knowledge of the plans is faulty or incomplete.

In fact, Pollack's treatment of plans as beliefs draws attention to the general fuzziness of the line between plan preconditions and domain assertions, where alternate formulations of the same domain might well encode the same fact in alternate ways. The number of courses required for a degree program, for example, could either be included directly as part of a **get-degree** plan in the precondition **(courses-required 15)** or it could be represented as an assertion about the state of the world, **(courses-required ?degree-program ?course-count)**. The allocation of data to plans or to world assertions, therefore, is always somewhat arbitrary, although the former choice implies agent knowledge while the latter leaves that question to the agent knowledge model. Thus the precondition/assertion distinction embodies useful predictions about the agent's knowledge, with the data coded as plans being that which is more stable and basic to the domain, and which thus can reasonably be assumed to be part of the agent's initial knowledge. The model presented here follows this traditional and useful distinction between plan and world knowledge, assuming that the knowledge coded as plans is part of the agent's initial knowledge. While relaxing that assumption would allow plan elements to be questioned, a strong heuristic bias would then need to be encoded representing the much greater likelihood of the agent's querying assertions rather than preconditions, making the resulting model not as different in practice from the one presented here as might at first appear.

The central concern in this section, however, is not with the standard metaplan model's assumption about the agent's full knowledge of the plans in the domain, but with extending that model's assumption that the agent does not know any of the facts in the domain by adding heuristic judgments based on the relative likelihood of the agent's knowledge of the different sorts of facts that are coded as assertions. Note that the metaplan model itself lumps all such non-plan data together,

and makes no assumptions about the comparative likelihood of the agent's knowing the different facts. The metaplan model itself cannot depend on the agent's level of world knowledge because the expert has no way of knowing in general which preconditions the agent does know and which not. Since the expert's best indications in this setting about the agent's world knowledge are usually only probabilities, their influence on our model of problem-solving actions is as heuristic rules, rather than by being built into the metaplans themselves.

7.2.2.2 World Knowledge Relevant to the Planning Model

We pointed out at the beginning of this section on world knowledge how a model of the agent's knowledge about the facts relevant to building the plan allows the expert to predict more easily which branches of the plan tree the agent will need to and choose to explore. In this and the following sections, the issue is identifying which classes of knowledge about the agent's knowledge are both most useful in controlling the growth of the metaplan tree and are also practically speaking available to the expert in the expert advising situation. The point is to select those classes of agent world knowledge that will be most directly useful to the heuristics presented in later sections.

Even leaving aside the plan knowledge, which we will continue to assume the agent knows and knows correctly, there are various classes of agent world knowledge which could be modeled and which would be helpful in predicting the agent's problem-solving moves, but the Pragma model only attempts to capture the agent's knowledge of assertions that are directly relevant to the plan feasibility core of its metaplanning model. Thus the heuristics currently only make use of a model of the agent's knowledge about those facts that are immediately relevant to the process of constructing the metaplan tree, either formulas that are preconditions to domain plans in that tree or ones that determine the existence of possible fillers for open slots in those plans.

There are other kinds of agent knowledge not modeled here that would also, though less directly, affect the building of the plan tree by causing the agent to prefer one plan branch to another. The major such class of interest here is the evaluative facts that agents use in assessing the comparative cost of different plans and in choosing the optimal one. The only approach to capturing these evaluative facts in the model is their use through the ***evaluate-plan*** subtrees in predicting possible evaluative queries. To allow for that prediction, sets of assertions known to have important evaluative effects are stored with each of the domain plans, as described in Section 6.6. However, a full representation for such evaluative world data would require capturing what the implications of the different factors are, and what algorithm is used to compare and combine them, rather than just their current database values. Not only would this substantially complicate the model in general, it would also greatly increase the amount of knowledge that the system would require about each particular agent in order to be able to make any predictions about likely problem-solving moves. Fully modeling the agent's evaluative process would require a model not only of the agent's knowledge of the facts of the domain, but also of the cost functions the agent is using to assign values to the various outcomes, something that varies more from agent to agent than the fact and plan model does and that thus is even less likely to be discernible to the expert in the expert advising setting. Since the metaplan model of

plan evaluation itself does not yet support that level of detail, it does not make sense to try to model such facts in the world knowledge model.

7.2.2.3 World Knowledge Not Requiring Deduction

Another kind of agent knowledge that is not modeled is knowledge whose bearing on the current goals and plan tree is conditional upon deduction from other principles or upon combination with other interacting goals and plans. The system only models the agent's knowledge of the literal preconditions that appear directly in the active plans. While some facility for dealing with the influences of implied or contingent facts is clearly an important extension to the system, such additions will also greatly increase the complexity of identifying agent world knowledge effects on metaplanning, so that it will never be feasible to recognize all the deductive implications of a given piece of agent knowledge.

One example of agent knowledge that does require outside facts to establish its relevance to the current planning situation comes from assertions governing the agent's other plans and goals, some of which may interact with the current goal in a way that affects the likelihood of the different possible problem-solving actions. Because of differences in interacting plans, the cost functions of planning agents can be different even if they are confronting the same current situation with the same knowledge and goals, since the impacts of this plan choice on other unrelated plans and goals may be different. Thus a naval planner may in fact choose to send a distant ship to replace the damaged one if she is aware of another goal to build up the fleet in that area by transferring vessels from other areas, since the actions for this other goal overlap those for replacing the damaged ship so that the action's comparatively high cost from the perspective of the **replace-ship** plan is made up for by its benefit in the **balance-forces** context. Because our planning model currently has no way of representing these possible interactions between the focused plan and the user's other plans and goals, it does not attempt to account for other plans that the agent might be involved in, and which she might be trying to coordinate with the current plan. Neither, therefore, does the heuristic component attempt to capture the agent's world knowledge about all the other plans that might interact with the current plan and thus affect the agent's evaluations.

The choice here to restrict the modeling of the agent's knowledge to classes of knowledge that directly affect the building of the plan tree was made both because that kind of knowledge was more directly useful in the heuristics and because it was easier to set up a model for the agent's knowledge of assertions that are already known to be relevant. As the metaplan model itself is extended to cover interacting plans and to model finer degrees of preference in place of binary feasibility for particular plans, it will be possible to make heuristic use of models of the agent's knowledge that support those features. For example, if there are two errands like buying milk and going to the library that could either be executed separately or combined, it might be that a strong preference for getting the milk home quickly might overcome a weaker preference for not making an extra trip downtown. The result of such an expanded heuristic system would be that the order in which plan branches would be explored would be sensitive to the relative preferences assigned to the combined outcomes.

7.2.2.4 World Knowledge Available in the Setting

Even as it becomes possible to extend the power of the problem-solving plan model to handle phenomena like evaluative choice between plans and the influences of one plan on another, there will still be the question of whether the setting in which such an expert operates would provide enough information about the agent's knowledge in these more contingent areas to be useful heuristically in directing the plan modeling. There are many settings in which no such detailed data about the agent's knowledge is available, so that only general and direct heuristics of the sort presented here can in fact be used.

Of course, even a greatly extended metaplan and agent's knowledge model still would not allow the expert always to predict exactly which paths the agent would explore in which order, both because it will not be possible in general to insure that the expert's model is enough like the agent's in the weights that it assigns to the different factors to be sure that the same branch would be scored highest in both models and because human agents do not always behave logically. The eventual goal, therefore, is not necessarily exact prediction, but the kind of smooth and effortless plan tracking leading to cooperative problem-solving responses that we see in human expert behavior, where the amount of effort expended in prediction is efficiently tuned to take maximal advantage of the data available about the agent's plans and knowledge in order to maintain enough of a model to steer the expert's responses in the most helpful direction. It even seems that human experts adjust the effort devoted to plan tracking based on need, so that more effort will be expended, even with poor data about the agent's goals, if there is a choice of plan matches that strongly affects what response the expert should give, while even strong data about the agent's goals and knowledge need not be followed far if the plan path is obvious without it.

Although that sort of powerful and flexible model of the agent's knowledge is the eventual goal for supporting cooperative interaction, the nature of the expert advising setting, as mentioned before, places severe limits on the amount that the expert is likely to know about the agent's knowledge, since the interactions are assumed to be fairly brief ones in which the expert does not have time to develop a detailed model of the agent. Thus it is not an unreasonable strategy in this context to settle for a simple model of the agent's knowledge and for straightforward heuristics applying that model to controlling the search in the plan tree for links to the succeeding query.

7.2.2.5 The Agent World Knowledge Model

We have already shown some of the complexities involved in determining which portions of the agent's world knowledge are relevant and in acquiring a model of the agent's knowledge in those areas. The particular model developed for Pragma relies on a combination of two approaches for modeling agent world knowledge, a default model acquired initially with the domain knowledge containing assertions of belief and disbelief that are expected to apply to all agents, and a set of particular assertions about the current agent's world knowledge, assertions that have been acquired in the discourse so far and that augment or supplant the assertions in the default model as appropriate.

The default model is intended to to describe certain facets of the typical agent's knowledge of the domain. Thus, assertions in the actual description of the state of the world can currently be marked as ones that the typical agent can be expected to believe are true, false, or unknown (the default). The set of those assertions with non-default values in the agent knowledge model describe the beliefs about the state of the world that can be assumed as the "common knowledge" of the great majority of users. As we will see, if one of these "common knowledge" assertions directly contradicts a precondition of a plan that the agent might otherwise be supposed to be considering, then the heuristic component will downgrade exploration of that branch in the plan tree. Similarly, heuristics based on "common knowledge" assertions suppress the exploration of whatever node in the plan tree would otherwise generate a query subtree modeling a query from the agent to determine the truth of that assertion, since the system assumes that it is already known.

In fact, the default model is implemented by means of a binary "commonly known" flag attached to each assertion used as a precondition in any plan. If that flag is set, it means that the logical status of that assertion, whether true or false, should be considered to be common knowledge, while the flag not being set means that the system makes no assumption about the agent's knowledge of that assertion. The advantage of this encoding is that the agent knowledge model does not need to change when the status of the assertion in the database changes; the disadvantage is that there is no way to represent the anomalous situation where all agents are assumed to believe that an assertion is true which is in fact false or vice versa, but it seems that that situation can safely be ignored. It it still quite possible in this model, of course, for particular agents to have incorrect beliefs about assertions assumed to be commonly known, with the result that the heuristics will be misleading in those cases.

While the default model gives some control over the search space, considerably greater power would be achieved if the system could maintain a separate model for each agent's knowledge, rather than just relying on default tags attached to the actual state of the domain as the best predictor for it. As a first step in that direction, the system does record the current agent's beliefs about particular assertions as they become clear in the course of the problem-solving dialogue. There are some cases where the agent's knowledge can be predicted easily, based on facts the expert has just revealed in previous answers. For example, where the default assumption may first have been that the agent was not aware of a fact that makes a particular subplan infeasible, a previous query and response may give evidence that the agent does now know that fact, and thus is likely to abandon this branch of the planning tree.

The following is a more detailed example of such a heuristic, where the previous dialogue establishes new data about this particular agent's world knowledge that in turn is used to control the search of the plan tree. When the previous interchange involved an ***ask-pred-value*** query tracked as arising from one of the preconditions in a particular domain plan and the expert's answer to that query had been negative, that answer adds a new element to the agent-specific world knowledge model recording that this agent now knows that that precondition is false. That new datum in turn affects the heuristic weights assigned to the plan tree nodes and used to control the search for a match to the agent's next problem-solving move, since it implies that the agent now knows that the current plan is not feasible in the given situation. Thus, while the default probabilities expect the agent to explore subplans

and subactions of the current plan class first, in this case, that becomes quite unlikely, and the alternative strategy of considering other related plan classes, ancestors of the currently focused one, or their subplans or subactions correspondingly more likely.

Note that the agent knowledge model does not at the moment trace knowledge deducible from other assertions in the agent knowledge model. This includes even simple cases of the direct inverse of an assertion whose truth is known, so that it is possible in this model to represent an agent who knows that the readiness of the Frederick is C1 but does not know that its readiness is not C2. The plans involved in test cases so far have not been complicated enough to require such a deductive model of agent world knowledge, but it is clearly an area for further work to extend the modeling of agent world knowledge to capture cases where the state of the agent's knowledge can be deduced either directly from other facts about which the agent's knowledge is known or by using them in combination with knowledge of dependencies in agent world knowledge.

7.2.2.6 Effects of Agent's World Knowledge on Subplan Growth

In this and the following sections, we cover the different areas of metaplan tree growth where the heuristics can make use of this model of the agent's world knowledge to affect the weights for different exploration paths in the tree. In this section, we examine the impact of the agent world knowledge model on subplan and subaction branching. In the following sections, we will cover its impact on query generation and variable instantiation. The first class of heuristics based on the agent's world knowledge therefore deals with the effects of the agent's knowledge of the truth or falsity of particular instantiated domain assertions on the growth of subplan branches in the plan tree. In this section, we show how a model that can predict which assertions the agent knows the truth of and which the agent does not know allows the expert to tailor the growth and search of the metaplan tree used for plan tracking, avoiding the exploration of branches that the agent will not consider and focusing on those the agent is most likely to explore. The actual heuristic rules, as pointed out in the previous section, are based for the most part on predictions about typical agents' knowledge of classes of assertions, and they use those predictions to affect the weights assigned to those metaplan tree nodes whose probability is somehow dependent on the agent's knowledge of the particular domain fact.

In spite of the limitations on the information about the agent's knowledge available in this model, important heuristic guidance can be obtained even from this partial model of the agent's likely knowledge about the assertions in the domain that are preconditions to the plans being considered. The main heuristic payoff is that wherever a subplan contains a precondition that the agent can be assumed to know is false, we can safely predict that the agent will not bother actively exploring that subplan. Examples of this include both whole ***build-subplan*** branches ruled out by some factor derived from the problem goal itself and particular ***instantiate-var*** branches ruled out by some feature of the instantiating entity.

One example of the first type would be where an entire ***build-subplan*** subtree would be ruled out due to some domain facts; in the damaged ship domain, an example would be where the **supply-by-helicopter** subplan of **supply-replacement-**

parts has a precondition that the receiving ship (already instantiated in this context as the damaged ship) must have a helicopter landing pad. (Assume that the damaged ship is far enough from the helicopter's base that the chopper must land and refuel, rather than just dropping the part from the air.) A person planning for the repair of a ship known to be too small to land a helicopter would not bother exploring other details of that subplan branch.

In that example, the fact that blocked the subplan tree was a direct precondition of the domain plan at its root; in general, the infeasibility of a subplan tree may be due instead to a lower precondition, or to an arbitrarily complex combination of precondition blockages at various levels. The heuristic rules given here, however, ignore these more complicated cases, and will only block exploration of a subtree when there is an immediate precondition of the domain plan at the node currently being explored that the system believes the agent knows will fail.

The h-rule form for examples like this is shown in Figure 7.6.

```
(def-h-rule block-known-failure-build-plan (?node)
  conditions ((node-ps-plan-name ?node build-plan)
              (node-domain-plan-name ?node ?plan)
              (domain-plan-precondition ?plan ?pred)
              (known-to-be-false ?pred))
  actions ((dec-score ?node 80)))
```

Figure 7.6: Block-Known-Failure H-Rule

Its effect is to severely decrement the score of any ***build-plan*** node whose domain plan as currently instantiated contains a precondition that the expert's model predicts that the agent believes to be false, whether that prediction is due to the common knowledge flag associated with that assertion or deduced from the earlier course of the dialogue.

Another example of knowledge of a domain fact ruling out a certain subtree of the plan tree would be where a particular variable instantiation creates a subtree with an immediate precondition of the domain plan at the root that is known to fail. In such cases, there is an h-rule that allows the system to downrate exploration of that branch right at the ***instantiate-var*** node, without needing to explore the ***build-plan*** node underneath it that would trigger the previous h-rule. The h-rule that catches these ill-fated ***instantiate-var*** nodes is shown in Figure 7.7.

```
(def-h-rule
    block-known-failure-instantiate-var (?node)
  conditions
     ((node-ps-plan-name ?node instantiate-var)
      (node-domain-plan-name ?node ?plan)
      (domain-plan-precondition ?plan ?pred)
      (known-to-be-false ?pred))
  actions ((dec-score ?node 80)))
```

Figure 7.7: Block-Known-Failure-Instantiate-Var H-Rule

Note that the only effects on subplan growth stemming from agent world

knowledge are restrictions that block off plan subtrees due to a precondition known to be false. While this seems one-sided, it is because the default assumption of the metaplan model is that the agent does not know the status of the world knowledge assertions relevant to the current plan, and that default assumption causes the system to search all possible branches of the plan tree, since all of them might seem possible to the planning agent. That search space can only be trimmed when the expert knows that the agent is aware of a failed precondition that blocks one of the subplan branches. On the other hand, there is no direct effect on tree exploration of a precondition that the agent is known to know will succeed. This point will be reconsidered, however, in Section 7.2.3 on predicting domain plan choice statistically, since there agent knowledge of successful preconditions can influence the default probabilities of particular subplan paths when a path that is usually blocked by some failed precondition and whose default probability is therefore low ends up being possible because that precondition this time succeeds, and, more to the point, is known to be known to the agent to succeed, so that the agent's knowledge of that successful precondition causes the probability of that normally unlikely subplan branch to be increased.

7.2.2.7 Effects of Agent's World Knowledge on Query Metaplans

A model of the agent's world knowledge affects the heuristic weights not only of subplan branches, but also of branches based on query metaplans. In these cases, data about the agent's knowledge of the world allows the heuristics to avoid exploration of plan subtrees where the subtree's metaplan is a query plan whose goal would be to discover some fact that Pragma already has reason to believe the agent knows to be either true or false. For example, in the naval domain, it might be that the basic status of the few aircraft carriers in the fleet would be an important enough fact that any user of the information system would be expected to be aware of it, while the status of the smaller vessels could easily have changed without provoking the uproar that would ensue if a carrier were suddenly rated nonfunctional. If that were true and the plan context were one in which the deployability of a carrier was one of the preconditions, the system could make use of the agent's assumed knowledge about that assertion to suppress the exploration of the *ask-pred-value* or *check-pred-value* nodes that could otherwise be built in searching for a match to such a query.

The h-rule for examples of this sort, where the agent is assumed to know that a particular precondition is satisfied, is shown in Figure 7.8.

```
(def-h-rule block-query-of-pred-known-ok (?node)
  conditions ((node-ps-plan-name ?node
                 (one#of ask-pred-value
                         check-pred-value))
              (node-queried-precondition ?node ?pred)
              (domain-plan-precondition ?plan ?pred)
              (known-to-be-true ?pred))
  actions ((dec-score ?node 80)))
```

Figure 7.8: Block-Query-of-Pred-Known-OK H-Rule

The effect here is to reduce the score of query nodes for assertions the agent is

expected to believe already to be true.

The examples that typically trigger this heuristic in the Pragma system are those like the above where the fact is a prominent enough one that it can be assumed that the agent is aware of its status, whether it is true or false. Note that there is a somewhat different sort of precondition whose truth the agent can be assumed to be aware of because it is simply very rarely false. In an academic setting, for example, there may be a *pro forma* requirement that the adviser approve all changes in course registration, but it may be so universally assumed that students weighing various courses of action do not consider the need for approval as a criterion to ask about. In such cases, too, this heuristic correctly models that agents are not likely to ask about such preconditions, since they have already assumed them to be successful.

The cases so far have all been examples where the precondition in question was known for one reason or another to succeed. The complementary case where the precondition is known to fail overlaps the subplan heuristics from the previous section, since in such cases the subplan heuristic will prevent that subplan branch from even being explored, so that the system will never get to the point of modeling possible queries about preconditions within that branch. Thus the main application for this class of heuristic will be in cases where the preconditions are known to be successful, when the metaplan model will explore the plan branch but this heuristic will prevent that particular precondition from being queried.

7.2.2.8 Effects of Agent's World Knowledge on Variable Instantiation Metaplans

This section presents a class of heuristics that exploit predictions of the agent's knowledge about the probable size of the sets of possible fillers for variable slots in the domain plans to control the weights given to the various possible variable instantiating metaplans. The primary effect is to adjust the likelihood of ***ask-fillers*** and ***constrain-var*** subplans depending on the agent's guess of the number of entities in the domain that fit the specified constraints. For example, if the agent expects a small set, she is more likely to ask for it to be listed in full, while if she expects a large set, she will be more likely to add some constraints to cut down the set size before asking for fillers. These heuristics thus test the actual size of the set in contexts with open variables, taking that as a heuristic measure in turn for the agent's guess at its size, and use that result to affect the relative weights given to ***constrain-var*** versus ***ask-fillers*** plans.

The rule in Figure 7.9 implements one such heuristic, reducing the likelihood of ***ask-fillers*** when the size of the resulting set would be larger than 30.

```
(def-h-rule reduce-ask-fillers-with-large-set (?node)
  conditions ((node-ps-plan-name ?node ask-fillers)
              (query-set-var ?node ?set-var)
              (> (cardinality ?set-var) 30))
  actions ((dec-score ?node 10)))
```

Figure 7.9: Reduce-Ask-Fillers-with-Large-Set H-Rule

The cardinality test applied to the ?set-var variable references the result of a

computation carried out and stored when the parent *build-plan* containing the open variable slot was first explored that tested in the actual database the number of entities satisfying the constraints on possible fillers of the slot and attached that data to the tree for heuristic use as a guide to the agent's sense of the probable size of that set. The effect of the rule is to somewhat decrease the score for *ask-fillers* nodes when the number of fillers is relatively large, with the cutoff here set to 30. A corresponding boost is given to *constrain-var* nodes in such cases by the rule in Figure 7.10.

```
(def-h-rule boost-constrain-var-with-large-set (?node)
  conditions ((node-ps-plan-name ?node constrain-var)
              (query-set-var ?node ?set-var)
              (> (cardinality ?set-var) 30))
  actions ((inc-score ?node 10)))
```

Figure 7.10: Boost-Constrain-Var-with-Large-Set H-Rule

These heuristics are like those in the previous sections in helping to predict the agent's problem-solving moves based on her assumed approximate knowledge of certain facts about the domain and the implications of that knowledge for which problem-solving strategies would be most useful. However, while those made use of an explicit model of which assertions in the domain model could be taken to be the common knowledge of all agents or specifically attributable to the current agent, these heuristics are based on a more general assumption that the agent has approximate knowledge about the entities in the domain that can be predicted from the actual facts of the domain, so that, for instance, while the agent might not know exactly how many destroyers are within range, still she is likely, judging from the total number or the number in the general area, to predict the value fairly closely. There are two reasons not to maintain as explicit a model in this case as we did in the case of individual assertions, first because it would be very expensive, since there are no individual assertions to index on, but rather arbitrary combinations of assertions given the particular plan context, and second because the agent herself, even if we had perfect access to her beliefs, is likely to have only vague knowledge to support these cardinality predictions.

The following list gives brief descriptions of other heuristics in this class that apply to the set instantiating metaplans and that use the actual world model to predict agent world knowledge. The actual h-rules for these heuristics are similar enough to the example already presented that they are not given in full.

- *Ask-fillers* metaplans are only likely when the candidate list is short (as in the **reduce-ask-fillers-with-large-set** h-rule given above).
- *Constrain-var* metaplans are more likely when the candidate list is long (as in the **boost-constrain-var-with-large-set** h-rule above).
- *Limit-cardinality* and *sort-set-by-pred* metaplans are also more likely when the list is long.
- *Ask-existence* ("Are there any...") queries are more likely when the list is short.
- *Ask-cardinality* ("How many...") queries are somewhat more likely when the list is long.

- *Instantiate-var* queries are slightly more likely when the candidate list is short.
- The existence of a single candidate (a cardinality of 1) encourages *instantiate-var* and discourages *ask-fillers*.

There is also some overlap between this class of heuristics and the following in that the queries being selected among by these heuristics can establish items of agent world knowledge that should be used in calculating the weights for other plan branches in the future. For example, after an *ask-cardinality* query, the agent knowledge model can be updated so that prediction of an *ask-existence* query for the same variable will be suppressed.

7.2.2.9 Using Agent World Knowledge Established During Consultation

The previous sections have presented different sets of heuristics based on assumptions about the agent's world knowledge that were established in general for all users before the problem-solving session began. We referred to the need for this to be expanded eventually to allow for an ongoing model of the knowledge of each individual agent, but pointed out that there is seldom opportunity for development of such an individual model in the expert advising setting. Nevertheless, there are cases where some facts about the agent's world knowledge become evident in the course of an advising session, either because the facts have been direct subjects of queries or because the course of the plan building allows the system to deduce the agent's knowledge.

The simplest example of this, of course, is where the agent explicitly asks about the value, which establishes from then on that the value is known. The Pragma system records for each assertion that is the subject of a query that the agent is now aware of it, and that fact will be used from then on in heuristically ranking the different search paths. For example, if a part-time student considering taking one course asks about the location of the building it meets in, the agent knowledge model would correctly predict that she would not need to ask the same question about a second course that she knew met in the same building.

One unusual example of this simple class comes when the assertion that the agent asks about is one that was misleadingly predicted to be one commonly known. That incorrect prediction will cause the system some trouble when it tries to match to the query about that assertion, since these very agent knowledge heuristics will postpone the exploration of the path that should match it, an analogue of the confusion people have in understanding and tracking a query that contradicts their expectations about the speaker's world knowledge. However, because the problem is with the predicted agent knowledge of that particular assertion, these heuristics will work against it equally wherever it appears in the plan tree, so that even though the desired match will not be found as quickly as it otherwise would, at least no other instance of the same assertion will match before the closest. There is in fact a secondary effect here that whenever the agent asks about a particular assertion, that in itself implies strongly that the agent is not aware of the value of that assertion. (We do not in this expert advising situation consider examples where the purpose of asking the question is

other than to discover the answer, even though such cases do arise in more complicated discourse situations, where queries are asked to draw the expert's attention to a fact that the agent already knows, or to mislead the expert as to the agent's knowledge.) This example is not useful for future prediction, however, since the same interchange that reveals the agent's lack of awareness also, with the answer, satisfies it, so that the effect of a query for future prediction is always to reduce the likelihood of that assertion being asked about again or of plans blocked by it being explored.

In addition to the simple cases where the assertion is queried directly, there are cases where a query about an assertion on one metaplan tree branch may reveal the agent's state of knowledge of another, for example, cases where the agent's lack of knowledge of a particular assertion can become clear in a more roundabout way. In cases where there is a precondition that the system judges is commonly known to be unsatisfied, it predicts that agents will not explore that particular subplan branch. But if, in spite of this expectation, the agent does ask a query that is suggested by that subplan branch and that only matches there, then the system can conclude that its original judgment about the common knowledge of the failure of the first assertion was incorrect. In practice, this would be implemented as a post-process to ***ask-pred-value*** that would check to see if any of the other preconditions of the queried plan branch were currently recorded in the agent world knowledge model as known to be false, changing their status to unknown.

Note that the Pragma system only takes account of new facts about the agent's world knowledge in the cases listed here where there is a definite implication. There are also cases where weaker, uncertain presumptions about the agent's knowledge could be derived from comparing the course of an agent's plan-building to the statistical expectations discussed in the following section. For example, if a precondition on an initially-probable subplan is in fact false (and unusually so, given the plan's general popularity), and an agent chooses not to explore that subplan or ask any queries tracked to it, the system might be justified in presuming with some degree of likelihood that the agent does know of the failure of that precondition, and that that is why she is not exploring that particular branch. Thus a business traveler asking about trains from Boston to Buffalo during a storm which has closed the Buffalo airport for some time may be presumed to be aware of the unusual fact of the failed precondition on the more typical plan of making the trip by plane.

The inverse case would also hold, where there is a generally unlikely branch due to a precondition that usually fails, but the agent explores it anyway, without querying the status of the surprisingly-valid precondition that makes that subplan possible, which the system could take as presumptive evidence that the agent was already aware of the precondition's truth. For example, a person who is stopped on the street at 8:00 p.m. on a Sunday shortly before Christmas and asked whether the department store on the next block would be likely to have a certain kind of sweater is probably justified in assuming that the speaker recognizes that while such stores would usually be closed at that hour, they are in fact open in that season.

Because such interactions between the agent world knowledge model and the statistical predictions of likely plan explorations are necessarily more tenuous than either heuristic alone, a full model of their effects on the probabilities of the separate plan branches would be quite complex, and would require a sophisticated system for

weighing probabilities and partial knowledge against each other. Thus the Pragma system only models that fixed set of situations in which the course of the interaction makes clear the agent's knowledge of the status of a particular assertion without simultaneous reference to the statistical model.

7.2.3 Predicting Domain Plan Choices Statistically

The previous parts of this section have described classes of heuristics that are based first on the shape of the metaplan tree and the particular patterns of metaplans in the area being searched and second on the agent's world knowledge of assertions that influence the likelihood of her considering particular plans. The first set therefore helps to predict the agent's problem-solving style of moving around in the tree, and the second predicts how the agent's choice of plans and queries is influenced by what her world knowledge indicates is feasible. In this section, we consider a third class of heuristics of a rather different sort that help to predict the agent's choice among the set of domain plans that are all apparently feasible. It is often true, for example, that a given plan class node will be realized much more frequently by means of one subplan class than another. For example, most students pursuing an **earn-credit-in-course** plan do so by means of its **take-course** subplan rather than by means of **transfer-credit**. Therefore, that path is the sensible one for the expert to explore first in predicting likely follow-on queries in an **earn-credit-in-course** context. Because the system does not have access to a detailed enough model of the agent's goals and preferences to predict the most likely choices of each particular agent, a statistical model of subplan and subaction feasibility collected from analysis of a sampling of typical problem situations is used instead.

The choices modeled by these statistical heuristics are those on which the more direct methods from the previous section have no bearing because no common knowledge assertion exists to make the distinction. For example, the **supply-spare-part-by-helicopter** plan depends on the presence of a landing pad on the damaged vessel, an assertion whose truth or falsity is marked as common knowledge. Therefore, if that assertion is in fact false for the actual class of vessels that includes the damaged one, the expert can predict that the agent is likely to know that, and, since we assume that the agent is aware of all the preconditions for the different plans, the expert can also predict that the agent will not bother exploring that path. However, consider the difference between the plans **supply-spare-part-by-frigate** and **supply-spare-part-by-battleship**, both subplans of **supply-spare-part-by-vessel**. There is no commonly known precondition to either plan that blocks its consideration; that is different from saying that both are definitely feasible, since there may in fact be no vessel of that type able to make the supply run, but at least neither one is *prima facie* impossible. Thus the agent knowledge heuristics as formulated in Pragma are not able to distinguish between the two. But there is nevertheless a major practical difference, because one is much more likely to use a small ship like a frigate for a spare part supply run than to redirect a battleship with all of its support vessels. Yet the reason for that is due to the comparative costs of the two options, rather than to any inherent impossibility. One option for capturing that difference would be to model directly the cost function for the different alternatives, but that would require a much more complicated domain model than Pragma at the moment supports. Therefore, the current approach is to use statistics collected from previous experience with the plan

tree as a guide to which plans are likely to be preferred by agents in cases where no common knowledge assertion distinguishes between them.

In earlier work [42], we suggested and explored a way of using statistics gathered across many uses of an expert problem-solving system to derive at each node the relative probabilities of success for each of its children. The trees in that work were AND/OR solution trees representing boolean combinations of preconditions (AND nodes) and alternate plans (OR nodes), and the statistics in that system were collected by testing the full tree of possibilities against the world situation of each previous user, recording at each tree node its success or failure given that situation, and thus collecting statistics that described the general probability of success at each node. These probabilities were used to predict the agent's likely focus of attention in searching for a solution subtree in a new situation. Where there were alternate plans (OR nodes), the heuristic expectation was that the agent would be likely to explore first the branch that had the greatest average likelihood of success, so that a high statistical success rating there predicted also a high chance for initial exploration. On the other hand, where a node involved an AND combination of preconditions, it seemed that the agent's motivation was to explore first the path least likely to succeed, since a single failure would mean the entire node was infeasible, and starting with the branch most likely to fail tends to minimize the time wasted exploring nodes that will eventually fail anyway. Thus for OR nodes, it was a low statistical success rating that predicted a high chance for initial exploration.

With some adjustments, the same statistical approach could be added to the heuristic component of the Pragma system, using the ***build-subplan*** nodes as OR branches and ***build-subaction*** nodes as ANDs. Because an expert advising interaction does not usually explore the entire tree, the statistics would need to be collected by applying the metaplan tree for known top-level goals to a sequence of world models representing various realistic situations, and allowing the feasibility judgments to percolate up the tree. Note that because the domain plan classes for the metaplan tree are in part dependent on the world state since that is what determines which sets of action sequences will share certain sets of effects, this derivation of statistical scores would be dependent to some degree on the problem situation, but given the generic plan classes usually of interest, this does not pose a significant problem.

The feasibility probabilities thus derived would add a finer degree of heuristic control than can be accommodated in the common knowledge approach. They would implement the heuristics that an agent when presented with a choice of plans will tend to explore first the plan that is most likely to succeed, while when considering a plan with a set of required actions, will explore first the action which is most likely to fail. For example, in the damaged ship context, **replace-spare-part** will presumably be the subplan of **restore-damaged-ship-readiness** that has been employed most often in previous problem-solving at this node, so that the expert can predict that that is also the most likely subplan for the agent to explore first in this case. On the other hand, within the **replace-spare-part** plan, the action for **transport-part-to-vessel** may be the one that most often fails, so that once the agent has selected that plan for exploration, the expert can expect her to test the feasibility of that action before testing others

An alternative possible use of heuristics that might be explored in this context, taking advantage of the richer context model available in Pragma, would be one that recorded information about not just the problem situations of previous users, but also about their plan exploration patterns. In that case, the statistics to collect would be ones reflective of the subplans and subactions that agents have tended to explore first, rather than those that would have turned out to be feasible if explored. Such statistics would take longer to collect, but would be able to handle a situation where there is an undesirable but almost always feasible solution whose frequent feasibility suggests its early exploration while its undesirability means that agents in fact postpone consideration of it as long as possible. Walking home from school might be an example, where rides or buses would be first considered, though less often feasible. Statistics based on the actual plan exploration patterns of previous agents would allow such preferences to be more directly modeled.

However, either of these approaches to statistical heuristics suffers from needing to judge automatically which sets of situations are closely enough related that their statistics should be suggestive for each other. With the statistical feasibility heuristics, this arises in the picking of situations to derive the scores, while in the exploration pattern cases, it comes up in knowing how to adjust the pattern data to account for the different situations faced by the different agents. A more powerful method than either one lies in expanding the effective modeling power of the metaplan model to handle the factors which are causing the plan preference so that they can be modeled directly and explicitly in the metaplan model and in the agent knowledge heuristics, as the feasibility criteria now are. That approach seems to offer in the long run a more adequate and useful model than any based on statistics of previous runs.

7.3 Heuristic Control from the Query Parse

All of the heuristics in the previous section (Section 7.2) are based on features of the pragmatic context, whether details of the metaplan tree structure, agent world knowledge from previous utterances, or statistical measures of the plans in previous use, all of which are independent of the actual query that the system is currently trying to match. In this section, we discuss the kinds of heuristic guidance of the search that are available by using instead the description of the query to be matched. The basic approach, of course, is to boost the scores of branches in the metaplan tree that do contain elements needed for a match with the query, and suppress those that comparison to the query shows will not be able to match.

The one example in the system of a positive heuristic of this type is one which boosts the score of an ***instantiate-var*** node when the value to which the free variable is being instantiated is one that also occurs in the partial representation of the query. For example, if the query was

Where is the Fox?

the **replace-ship** node in the tree for replacing the damaged vessel could spawn many ***instantiate-var*** branches, each representing the consideration of a different possible value of **?rship**. The particular branch which explored the instantiation of **?rship** as the Fox would receive a substantial boost in heuristic score from this rule. Not only does this represent the greater probability of a match for that branch compared to its

siblings, which instantiated **?rship** to a value that does not occur in the query, but it also models the generally increased interest for the search as a whole in a branch that has discovered a slot in the plan that may explain the occurrence of that given value in the query.

The other uses made of information about the query being matched are examples that cut off search paths that can be shown to have no possibility of matching to that particular query, based either on the query operator or on the set of assertions available. For an example of the former, a query like

What is the course of the Vincent?

is represented using an **ask-value** query operator, and that operator is only introduced into the metaplan tree by the ***ask-pred-value*** metaplan. Since the query-generating metaplans are mutually exclusive, with only one occurring on any one branch in the tree, the other query-generating metaplans like ***check-pred-value*** and ***ask-fillers*** need not be expanded in this particular search, since they will certainly not provide a match. In the same way, we can see that the query-generating metaplans do not in themselves contribute assertions to the pool that can match those found in the query. Thus, the system can test for the possibility of a match between the ***build-plan*** node itself and the query descriptions. If the assertions from the query (ignoring the query operator) can be matched in the context of the ***build-plan***, then it makes sense to explore query-generating branches from that node, but if there is no match there, then the search can be cut short there, unless one of the ***constrain-var*** metaplans will be able to introduce the missing constraints. Thus a pre-match test at the ***build-plan*** node itself can provide strong direction to the possible fruitful query-generating branches at that node.

These heuristics that suppress plan tree branches based on information from the query are not coded as h-rules affecting node scores but are instead written directly into the tree exploration code, both because that method allowed easier coding of these fairly complex tests and because the results of these heuristics when negative effectively prevent further exploration of that branch, rather than just partially downrating it. (The justification for still calling these heuristics will be clear when we see the effect of applying them when there is only partial knowledge of the query form.) Nothing is lost in not factoring these into the heuristic calculation of the score of a node, since these heuristics only prevent exploration of nodes that would not be able to match with the given query, so that their ranking would never be an issue.

The two mechanisms that implement this negative search control based on knowledge of the query form in Pragma are called triggering and search matching. Triggering is the mechanism for applying knowledge of the query operator to filtering which of the query-generating metaplan nodes should be expanded. Each of the metaplans can have as an attached value a list of "trigger assertions" that are tested as the node is first being explored. The ***ask-pred-value*** metaplan, for example, carries the trigger assertion **(query-op ask-value)**, since a query modeled by ***ask-pred-value*** will carry that query operator. The trigger assertions are tested against the assertions describing the actual query being matched, with the result of preventing those query-generating metaplans that cannot match due to their query operator from even being expanded. Search matching is the other negative mechanism for applying knowledge from the query, referring to a pre-match performed at each ***build-plan***

node between the assertions present and inherited in that plan context and those requiring to be matched in the current query. If the result of this search match indicates that metaplan query branches rooted in that *build-plan* node could not match the actual current query, all the work of instantiating and testing for a match for that entire subtree is avoided.

The use of these mechanisms in Pragma is able to speed up the search in the metaplan tree substantially based on information about the query. Given the scope of the metaplan tree requiring search, the success of the system depended on making maximal use of such heuristic guidance both from the pragmatic context, predicting likely directions for the tree to grow, and from the current query, preferring directions that could lead to possible matches.

7.4 Linking to the Partial Interpretations of Ill-Formed Queries

While the previous sections have described the heuristics in terms of matching to a well-formed query by expanding and searching the metaplan context tree to track the intentions behind this new query, this section once again places this all in the context of using pragmatic knowledge to suggest resolutions for ill-formed inputs. With well-formed queries, the query representation could specify the query exactly, and the only question then was what locations in the metaplan tree could be shown to motivate that particular query and which of those (if there were more than one) would the heuristics select as the context most likely to be actually motivating the current agent in this follow-on query. However, with ill-formed queries, the representation can only partially specify the query, so that there will typically be many more places within the metaplan tree which can be linked to this partial representation than could be matched to the complete one of the well-formed query. (The term "linking" is used here to stress the difference between this and straightforward matching.) The presence of ill-formedness places correspondingly greater demands on the heuristic component, which now must sort out not just sites that match with the current query but all sites that can be linked to this partial specification of it, judging which of these multiple matches is most closely logically related to the previous context.

While the presence of ill-formedness expands the space of options that the heuristics must filter, the methods applied for examples of linking for ill-formedness are the same as those described in matching and tracking well-formed examples. The goal is to preserve as much as possible of the heuristic effect from the well-formed to the ill-formed case, so that the heuristic rankings can then be used not only to disambiguate multiple match sites in the tree but as part of the same process in the linking case to suggest a ranked set of possible fillers for the ill-formedness. In the rest of this section, we look briefly at the effects of the presence of ill-formedness on the heuristic component and how the existing heuristic approaches serve this new purpose.

There are two extra sources of ambiguity that must be dealt with in cases of ill-formedness, that introduced by the wildcards in the partial interpretations of the ill-formed queries, and that resulting from the presence typically of multiple partial interpretations, stemming from different wildcard parses of the sentence. Both introduce additional ambiguity in the linking process that must be resolved by greater dependence on the normal heuristic rules.

Given a single wildcard parse for an ill-formed query, the primary additional ambiguity that arises comes from the fact that the wildcards that fill the slots in the partial interpretation of the input sentence, whose actual intended contents are unknown, can match with any element of the plan context. For example, a fully-specified query like

What is the speed of the Shark?

is restricted to matching to metaplan nodes where some plan variable has been instantiated to the Shark and where the speed of the filler of that role is known to be relevant to the particular plan. In our example context of planning to repair or replace a damaged ship, there are a limited number of roles for which the speed of the ship filling the role is one of the listed plan constraints, namely, the roles of replacement ship or of the ship to be used in supplying a spare part. However, if the query were ill-formed, like

What is the seed of the Shark?

so that there was no possible relevant parse without resorting to wildcarding portions of the input, even the single wildcard parse that results from the choice of the fourth word as the wildcard and that might be represented as

*What is the ***** of the Shark?*

could match to any plan tree node requesting any attribute value for a vessel instantiable as the Shark, a much larger set. The problem becomes even worse in cases where the wildcard element comes in the position of a plan variable requiring instantiation; for instance, if the example were

What is the speed of the Shack?

then the wildcard parse resulting from the choice of the last word as the wildcard

*What is the speed of the *****?*

could match with instantiations to any vessel at all that could play one of the speed roles in the plan.

The normal heuristics, heavily dependent on the pragmatic context tree rather than on the current query, are the main line of defense against the increased ambiguity introduced by ill-formedness. The only class of heuristics that is directly affected by the presence of wildcards in the input are the search control methods that make use of information about the current query in determining which branches to explore, and they are programmed in the obvious way to allow exploration of any branch that might match with any instantiation of a current query that contains wildcards. Thus, for the example concerning *the ***** of the Shark*, the search match that tests whether query-generating metaplans should be expanded beneath a given ***build-plan*** node would allow their expansion in this case, even though there is no match but only a possible link between the wildcard representation of the query assertions and the assertions present in the ***build-plan*** nodes for the various vessel attributes that will link to it. The search match test is thus partially disabled in the presence of wildcards, allowing more nodes to expand as query-generating subtrees than will in fact be able to link fully with the ill-formed input, but while the query related heuristics are weakened by the ill-formedness, those depending on the previous context in the metaplan tree still

suggest a ranking over the possible fillers that provides the best indication supportable by the available data as to the most likely link.

The other difficulty introduced by ill-formedness besides wildcard elements, of course, is the presence of multiple wildcard parses for a single ill-formed input. This can arise either when the wildcarding of more than one word in the string produces a valid wildcard parse or when a single wildcard location in the string leads to alternative wildcard parses, due perhaps to an ambiguity elsewhere in the sentence. The current approach to multiple wildcard interpretations of either type is to feed them each through the Pragma system, collecting the ranked set of linked contexts resulting from each pass, and then to merge the results based on the rankings assigned by heuristics. This approach, although cumbersome, allows the power of pragmatic context to be used in selecting between the possible wildcard parses, since the one chosen will be the one of the ambiguous set that discovered the closest link in the metaplan tree to the previous context. There are certainly cases where even this approach will select an interpretation not intended, or will leave a wide variety of closely ranked interpretations to choose from, but it represents a consistent approach to making use of information from pragmatic context, which is the strongest source of guidance available, to direct the ill-formedness resolution.

CHAPTER 8

PRAGMA IMPLEMENTATION DESCRIPTION

8.1 Introduction

As pointed out previously, some elements of the theoretical approach described in the bulk of the preceding chapters have not yet been realized in the current Pragma implementation. This chapter both characterizes the coverage of the current implementation, pointing out which portions of the theoretical framework are included, and also explains the structure of the implementation, describing when appropriate the nature of the tradeoffs in choosing the given implementation style. Because Pragma was intended as a research tool, these decisions were often made in favor of flexibility and conceptual clarity, even at the expense of speed.

While the description of the approach given in the previous chapters does include features not yet implemented in the running Pragma system, enough of the approach has been implemented to establish its feasibility and to enable conclusions to be drawn about the usefulness of this kind of model and of these sorts of heuristics. In overview, the current Pragma system implements the full family of plan-building metaplans and most of the query metaplans, except for part of the compositionality of complex set queries. The parallel structure of evaluative metaplans with the code to maintain the duplicate trees and to switch back and forth appropriately in those cases and the inform metaplans are not covered in the implementation. For the heuristics, the current system implements the heuristic rules based on tree shape and metaplan context as presented, using the node rankings that result to control the search in the plan tree through an agenda mechanism. The heuristic rules based on agent world knowledge, however, remain to be implemented.

In the rest of this chapter, we discuss in turn the implementation of each of the Pragma system components, describing the major choices made and the style of implementation adopted in each case, and ending with a brief description of the place of Pragma in the context of the full BBN Janus NL system.

8.2 Wildcard Parsing

While a wildcard parsing facility as described in Section 4.4 remains a desirable goal, it was not possible to achieve an actual implementation of this in the Janus context within the scope of this research effort. Therefore, the wildcard parses used as input to the Pragma system were derived from the parses of well-formed queries by substituting wildcards by hand for the elements of the parse derived from the word assumed to be wildcarded.

An initial attempt was made, however, using the Janus ATN parser, to verify our expectations as to the number of parses that would result from treating each word of an ill-formed sentence in turn as a wildcard. Since Pragma's linking algorithm needs to be run with each possible wildcard parse, its feasibility in practice depends on there being typically a small number of wildcard parses for a given ill-formed sentence. This test worked by defining a set of pseudo-words, one for each syntactic class known to the parser. To test for parses with a particular word position treated as a wildcard, each of these pseudo-words in turn was substituted for the wildcarded word and the resulting string fed to the parser. If a successful parse was found, that meant that the result of a true wildcard parser would have included that syntactic possibility in its wildcard parse for that word position. For example, the ill-formed query

Is the Fox it port?

cannot be parsed as it stands. If the word *it* is wildcarded, a parse is found in which that fourth word is identified as a preposition. That parse was identified in our test procedure when the pseudo-word representing the class of prepositions was tried in that word position. In this example, that word position is the only one that yields a parse when wildcarded.

In most of our tests, we found more than one position that yielded a parse. For example, the query

The vessels in group 2 was deployed to the Med.

would find wildcard parses both when *vessels* and *was* were wildcarded. The most frequent result in our initial tests was that 2 of the wildcard positions produced valid parses, and that number was seldom higher than 3, which was an encouraging sign for this approach, implying that syntactic constraints alone were strong enough to rule out many of the possibilities opened up by wildcarding each word position.

This initial test of wildcarding in Janus only made use of the parser's syntactic component. To include semantics in the test would have required a way to disable any caseframe tests involving the wildcard value and to be able, instead, to create some semantic representation describing the class of entities that would have passed the test. For example, the query

What is the vocation of Fox?

is syntactically valid, but semantically ill-formed. If *vocation* were wildcarded, the semantic component should restrict the possible fillers to things that can be properties of a ship, while if *Fox* was wild, the restriction there would be to entities that can have vocations. Although such semantic processing was not part of our wildcard test in Janus, adding semantics would have strengthened the result, since many wildcard parses that appear to succeed syntactically would fail when semantic filtering was added.

Thus, the results of that brief study argue that the number of wildcard parses would not on the average be large enough to prevent running the Pragma linking algorithm for each one and using Pragma's normal heuristic scheme for ranking the resulting links. Actually implementing wildcard parsing and connecting such a parser to Pragma to confirm this conjecture remains a research goal, and one that will

probably be dependent on moving to a parser implemented in a different formalism. In particular, as pointed out in Section 4.4, it seems that unification parsing would be a natural setting in which to implement this wildcard approach.

8.3 Atomic Sentence Set Logical Representation

Pragma's logical representation was picked to make matching easy, since each test for a link requires testing for a match between the logical form of the query and the logical form predicted by the context metaplan tree. The logical representation used in the rest of the Janus system, the World Model Language (WML) described by Ayuso and Hinrichs [4], is not well-suited to this purpose. WML is an intensional logic designed to be derived compositionally from the components of the surface text in the tradition of Montague semantics [39]. It is a rich and complex formalism in which matching would have been very difficult, so an elementary representation in terms of conjunctions of atomic sentences was adopted instead. That choice made it possible to turn the matching over directly to a unification-based prover and also simplified the incremental specification of query contents in branches of the plan tree. In addition, many of the other components of the system including the tree expansion procedure and heuristic rule application mechanism were also implemented using the prover, so that communication between the components was greatly eased. Naturally, a price was paid in terms of representational adequacy in replacing WML with this simplified logical form, but the range of example queries required for developing and testing the metaplan model and linking approaches were representable adequately in the simple form. The rest of this section discusses these choices in more detail.

The very richness of the WML form of intensional logic used in the Janus system argued against its direct use in Pragma. There are frequently ways of phrasing things in WML that are logically equivalent in terms of the result of the query but that appear wildly different, since WML retains some elements of the surface linguistic form. For example, the query

List the readiness of the Fox.

is represented as shown in Figure 8.1.

```
(bring-about
 ((intension
   (exists ?jx3 list
    (object.of ?jx3
     (iota ?jx4 readiness.value
      (overall.readiness
        (iota ?jx5 vessel (name.of ?jx5 "Fox"))
        ?jx3)))))
  time world))
```

Figure 8.1: WML for *List the readiness of the Fox.*

This WML requests the bringing about of the result of evaluating a particular intension at given time and world indices, that intension being the existence of a list whose object is the unique (**iota**) readiness value such that it is the overall readiness of the unique vessel named "Fox". However, the closely related query

What is the readiness of the Fox?

is represented as in Figure 8.2.

```
(query
 ((intension
   (present
    (intension
     (pred-to-set
      (lambda (?jx7)
        thing
        (equal
          ?jx7
          (iota ?jx8 readiness.code
           (overall.readiness
             (iota ?jx9 vessel (name.of ?jx9 "Fox"))
             ?jx8)))))))))
   time world))
```

Figure 8.2: WML for *What is the readiness of the Fox.*

The initial speech act operator is different in the two queries, as one would expect, and the *what* clause in the second is represented explicitly by the variable ?jx7 of type **thing** that is **equal** to the object of the query.

To be sure, there are cases where linguistic differences of that sort do have significance with respect to the pragmatic context generating the query. For example, query forms like *what is* that are requests communicate a more formal and respectful attitude than command forms like *list*, and thus are appropriate in slightly different discourse circumstances. In such cases, there might be benefit to including that sort of detail in the metaplan model, so that query metaplans would predict not only the logical content of queries justified in particular situations but also features of their linguistic form, and those predictions could then be related to the particular pragmatic context. However, such refinement is far beyond the current research frontiers for Pragma. Including such detail in the logical forms would merely have required building code into the matching component that was able to recognize the equivalence of the two forms for these purposes. Thus, there was a strong argument in favor of a more restricted logical form.

The choice of a logical form came partly from the engine chosen for the matching task. The simplest choice for that seemed to a unification-based prover similar to Prolog except that it returns the full set of matches found when there are more than one. Such a prover could be implemented quickly in Lisp, although its slowness did end up placing some limitations on the use of Pragma for demonstrations. One helpful feature added to this prover's design was the ability in cases where no match was found to print out the subformula that was failing with the variable bindings generated in the matching attempt so far, data that often proved useful in debugging the plan tree expansion and linking examples.

Another criterion for the design was that the elements of the query be able to be built up easily from the relevant domain plan and metaplan elements in the tree. For

example, a query like

Which C1 cruisers are within 100 miles of Sterett?

has the **readiness** and **vessel-class** of the cruisers supplied by the **replace-ship** plan, while the distance limitation is added by an ***add-scalar-constraint*** metaplan further down the tree. The simplest representation allowing for such assembling of query specifications was a simple conjunction of atomic sentences, collected from all the parent plan tree nodes. There are naturally some kinds of queries that cannot be directly expressed in this form, including disjunctions[2] like

List the cruisers or destroyers in the Indian Ocean.

and queries requiring internal quantification, like

Which ships filed less than 3 casualty reports last year?

One area for further work is to expand the logical coverage of this representation to handle these sorts of cases, making the appropriate extensions to the matching code and to the code that assembles the query from the plan tree nodes for matching.

Currently, however, the logical representation for each input query is as a set of atomic sentences, meaning simple, unnested propositions. The predicate names are taken from a set of domain predicates with known type constraints on their arguments, and the argument positions are filled either with domain constants or with typed variables. The types of both constants and variables are recorded in a hierarchical type system that includes both simple types and set types.

Besides a set of such assertions, the representation of a query also includes a query operator taken from the following set with a logical status similar to a WML speech act operator that describes what is being asked:

- (**ask-value** <var> <sentence>) where <sentence> includes <var>
- (**ask-truth** <sentence>)
- (**ask-set-fillers** <var>) where <var> is of a set type
- (**ask-set-cardinality** <var>)

This representation of queries as a set of simple propositions plus a query operator is derived from an original semantic representation of the query in the nested, intensional, WML form by projecting away the quantificational and intensional material. There are thus further distinctions that can be represented in the original logical forms but that are lost in the translated form, in addition to the disjunction and quantification examples mentioned before where the result set specified by the query is affected. For example, distinctions between given and new information, like that between the following two queries,

How many C1 ships are in the Indian Ocean?

---

[2]A rudimentary disjunction mechanism using a **one#of** function with special proof axioms was added to the prover for use in plan tree expansion tests, but it has not been used in query representation.

How many ships in the Indian Ocean are C1?

will not be maintained; the translated form for both will be that for

List the C1 ships in the Indian Ocean.

as given in Figure 8.3.

```
pred-list ((type ?ship (set-of vessel))
           (readiness ?ship C1)
           (location ?ship Indian-Ocean))
query-op (ask-set-fillers ?ship)
```

Figure 8.3: Representation of *List the C1 ships in the Indian Ocean.*

Distinctions in quantifier scoping are also lost. For example, while the query

Which ships have visited each port in the Mediterranean?

has two interpretations, one where *each* takes narrow scope, listing those ships that have each visited all ports, and one where it takes wide scope, listing for each port those ships that have visited it, the representation of this query as a list of atomic sentences shown in Figure 8.4 is ambiguous.

```
pred-list ((type ?ship (set-of vessel))
           (type ?port (set-of port))
           (location-of ?port Mediterranean)
           (visited ?ship ?port))
query-op (ask-set-fillers ?ship)
```

Figure 8.4: Representation of *Which ships have visited each port ...*

However, while this representation does thus project away some of the exact meaning of complex queries, the ambiguous forms that result still provide a query representation that can be matched against the predictions of relevant queries and used for ill-formedness correction. The relevance of the individual assertions in the representation can be tested against the pragmatic model's predictions even if the truth conditions of the query as a whole are not exactly maintained. Again, one direction for further research would be to expand the system's representational power so that it could model and match against those features in the intensional logic that it currently ignores, but that further power would be essential only for the rare examples where those features directly affect the choice of match in the plan tree or are directly involved in the ill-formedness.

8.4 Plan and Metaplan Implementation

The use of sets of atomic sentences as the logical form carries over into the representation for domain plans, since the heart of each plan is its list of preconditions, which is in exactly that form. For example, Figure 8.5 shows the definition of the plan for restoring the readiness of one of the assigned slots in a battle group. Each plan has a header with function name and arguments that describes the goal achieved by the class of domain plans described here. The plan can introduce further local variables in addition to its arguments, and the types of all variables are declared in a list of pairs of

```
(def-plan-class restore-slot-readiness
  header (restore-slot-readiness ?slot ?group)
  vars (?ship ?srdy)
  types ((?group battle-group)
         (?slot battle-group-slot)
         (?srdy readiness-code)
         (?ship vessel))
  preconds ((assigned-to-slot ?ship ?slot)
            (overall-readiness ?ship ?srdy)
            (not-C1 ?srdy))
  actions ()
  subclasses ((replace-ship ?ship ?slot)
              (repair-ship ?ship)))
```

Figure 8.5: The Restore-Slot-Readiness Plan Class

variable and corresponding type. The preconditions serve to bind the variables, in this case ?ship and ?srdy, and to define the common restrictions on plans of that class, here that the ship currently occupying the slot must not have the top, "C1" readiness rating. This plan class divides into two plan subclasses, one in which the damaged ship is replaced and one where it is repaired.

The replace-ship plan class in Figure 8.6 demonstrates a few other features.

```
(def-plan-class replace-ship
  header (replace-ship ?ship ?slot)
  vars (?ship-class ?ship-loc ?cur-rship-loc)
  free-vars (?rship)
  constants (C1)
  types ((?ship vessel)
         (?slot battle-group-slot)
         (?rship vessel)
         (?ship-class vessel-class)
         (?cur-rship-loc location)
         (?ship-loc location)
         (C1 readiness-code))
  preconds ((vessel-class ?ship ?ship-class)
            (vessel-class ?rship ?ship-class)
            (overall-readiness ?rship C1)
            (location-of ?rship ?cur-rship-loc)
            (location-of ?ship ?ship-loc))
  actions ((sail ?rship ?cur-rship-loc ?ship-loc)
           (assign ?rship ?slot))
  subclasses ())
```

Figure 8.6: The Replace-Ship Plan Class

In this case, in addition to the normal variables that are dependent on the instantiated values of the arguments, there is also a free variable not thus determined, here ?rship, which an agent employing any plan in this class may instantiate at will to any entity of the proper type that fits the preconditions, here, any C1 vessel of the same class as the

damaged one. Note also that this plan class is not further divided into subclasses, but that there are two actions shared by all plans in this class, a **sail** action to get the replacement ship to the place where the damaged ship was, and an **assign** action to administratively deploy it in that slot.

As for the problem-solving metaplans, because they are plan classes themselves, chosen to represent the agent's plan-building moves, they are represented in a form similar to that of the domain plan classes, although the preconditions in this case are tested against the domain plan library to determine, for example, subplan relationships among plans, and against the metaplan tree, rather than against the world knowledge model. The definition in Figure 8.7 of the ***build-subplan*** metaplan, which encodes the step of building a plan in one class by picking one of its subclasses and building a plan included in it, will give the flavor.

```
(def-ps-plan-class *build-subplan*
  header (build-subplan ?plan-class-name
                        ?arg-values
                        ?free-var-values)
  vars (?sub-class-name
        ?sub-arg-values
        ?free-var-bindings)
  free-vars (?sub-class-header ?sub-free-var-values)
  preconds
    ((sub-class-header-of ?plan-class-name
                          ?sub-class-header)
     (car sub-class-header ?sub-class-name)
     (instantiated-plan-args
        ?plan-class-name
        ?arg-values
        ?free-var-values
        ?sub-class-header
        ?sub-arg-values
        ?sub-free-var-values))
  actions ()
  subclasses
    ((*build-plan* ?sub-class-name
                   ?sub-arg-values
                   ?sub-free-var-values)))
```

Figure 8.7: The *Build-Subplan* PS-Plan Class

The metaplans are called "ps-plans" because they encode problem-solving actions, and their names are by convention surrounded with asterisks. The first few arguments in the header of each metaplan are used by convention to carry the plan tree context, expressed as a domain plan class name, the set of argument values for this particular instance of that plan class, and the free-var-values if any specifying any instantiations of free variables modeled by higher ***instantiate-var*** plans. (No explicit type checking on arguments is done at the metaplan level.) The free variables in this case are the **?sub-class-header**, the instantiated header of the new domain subplan, and its **?sub-free-var-values**. (Note that the **?sub-class-name** alone does not determine the

?sub-class-header, since a plan class may have multiple plan subclasses that differ only in argument values.) The preconditions here test facts from the plan library like **sub-class-header-of** and facts computed as part of plan tree expansion, like the **instantiated-plan-args** form that stores the argument values for the subplan that were precomputed when the parent ***build-plan*** was first reached during tree expansion. The ***build-plan*** metaplan is the one subclass of ***build-subplan***, invoked now on the subplan.

The metaplans already implemented within Pragma include the full range of plan-building metaplans as described in Section 6.4 and almost the full range of query metaplans from Section 6.5; the evaluative and informing metaplans remain to be implemented. The plan-building metaplans include the ***build-plan***, ***build-subplan***, and ***build-subaction*** triad that outline the structure of the domain plans in the metaplan tree, and the metaplans involved in instantiating open variables, ***instantiate-var*** itself for binding a variable to a particular value and the ***constrain-var*** metaplans ***add-boolean-constraint*** and ***add-scalar-constraint*** for restricting the value of an open variable without yet binding it to a particular value. The query metaplans include the plan feasibility queries ***ask-pred-value*** and ***check-pred-value*** that ask directly about preconditions and the query metaplans related to the set of possible fillers for an open variable slot, like ***ask-existence***, ***ask-cardinality***, ***ask-fillers***, and ***ask-attribute-value***. Thus a large enough core of metaplans is implemented to demonstrate the usefulness of the metaplan approach for discourse modeling in this setting.

8.5 Domain Plan Library, Database, and Type System

Collections of domain plans in the form given above were worked out for each of the two situations within the naval deployment domain in which examples were developed, the damaged ship setting and the SPA prosecution setting. (These settings are described in Sections 9.2.1 and 9.2.2.) In each case, there was a single top-level goal that the agent was assumed to be pursuing which set the context for all the queries in the dialogue. Because the two settings fell within the same domain, many of the lower-level plans like those for sailing a ship from one location to another were shared by both settings, and could appear as nodes in either tree.

The preconditions in the domain plans were tested against a database of facts about the current world state represented as a set of atomic assertions. This database was populated with sufficient ship names and related data to ensure that suitable candidates would be available for the open slots in the plans. The root plans for each setting included arguments specifying, for example, which ship was damaged, and the database facts were naturally tailored to match those situation definitions.

Along with the database of domain facts there was a type system in the form of a collection of type assertions for each entity in the database and a set of subtype assertions that created a taxonomy of types. Thus, for the Fox, there was an assertion of its type, **(type-of Fox aegis-cruiser)**, and assertions relating that type to its supertypes, like **(sub-type aegis-cruiser cruiser)**, and **(sub-type cruiser surface-vessel)**. There were also deductive axioms allowing the unification matching code to conclude new type facts from the **type-of** and **sub-type** assertions, for example,

in this case, (type-of Fox surface-vessel).

When the system was operating in its instantiation mode, the plan preconditions and type assertions were tested against the database and type system as each new node in the plan tree was constructed. If no match could be found in the database for the instantiated preconditions, that domain plan was considered to have failed, and the node was ignored. If one or more matches were found, those values were used as the variable instantiations in new plan tree nodes. When the system was not running in instantiation mode, of course, a single instance of each plan node explored was built, never none or more than one, with the dependent variables bound to assumed, newly-generated constants, rather than to actual values from the database.

8.6 Building the Metaplan Tree

Given the implementation of the metaplans and domain plans in a form amenable to the unification prover, building the metaplan tree becomes a fairly straightforward process. Each metaplan node that has either metaplan subplans or subactions, beginning with the root node supplied by the situation, has the preconditions of those possible children suitably instantiated given the current bindings of any metaplan variables tested against a database of metaplan facts describing the metaplans and the tree so far. A child node is built for each successful match found in that database for the child's preconditions. For example, the ***build-subplan*** metaplan, which is itself a subplan of ***build-plan***, is instantiated once for each domain subplan of the domain plan in the ***build-plan***, assuming that the metaplan preconditions succeed in each case.

Metaplans like ***ask-pred-value*** do not involve moving to a different domain plan from the parent, while metaplans like ***build-subplan*** do. One of the preconditions of metaplans that do involve moving to a different domain plan is one that determines the variable bindings for the new plan. The assertions against which these preconditions match are created in advance by special code that precomputes when a metaplan node for the parent domain plan is first explored the variable bindings that would apply for each of its subplans or subactions, given that its own variable values are known. When the system is running in instantiated mode, this precomputation step is where the domain database is consulted, and 0, 1, or many instances of these assertions of precomputed values will be created depending on how many matches are found for the child's preconditions in the domain database. To adapt the algorithm for the non-instantiated mode, that precomputation step is altered to assert instead a single instance of the child plan with dependent variables bound to newly generated unknown constants.

Some alterations were made to the straightforward tree expansion algorithm by adding tests to prevent the exploration of branches that it could be predicted in advance would fail. Because these tests depended on the query being matched, they were discussed in Section 7.3, which covered heuristic search control techniques based on information about the query. They were implemented by means of triggering assertions added to particular metaplans, which were interpreted as additional preconditions on the creation of nodes based on those metaplans. For example, a trigger was attached to the ***ask-pred-value*** metaplan that tested whether the query

operator in the current partial parse was **ask-value**. There would be no chance of finding a link below that ***ask-pred-value*** node when that trigger did not fire, so adding the trigger did not change the result of the computation, but it did substantially decrease the size of the tree that needed to be searched. In this example, with similar triggers added to each of the query plans that maps directly to a query operator, the effect was that each queryable precondition in a ***build-plan*** node expanded only to a single query node, rather than to one of each type. Similar triggers were also used to control query branch expansion based on the predicate being queried and to choose between the two subplans of ***instantiate-var***, ***pick-value-suggested*** and ***pick-value-at-random***, based on whether or not a constant of the appropriate type to serve as an instantiation for the open variable occurred in the partial interpretation of the query.

Another efficiency move made during system development was to break up the database of assertions about the plans and about the tree structure against which the metaplan preconditions are tested, attaching the relevant assertions to each individual node in the tree, and then collecting the actual list of assertions to be fed to the prover from the neighboring nodes of the node being tested. Again, this merely sped up the prover by shortening the list of assertions which it had to search for matches without changing the result computed.

8.7 Heuristic Component Implementation

The heuristic rules described in Chapter 7 are used both for directing the search of the metaplan tree and for ranking the link nodes found. In their former role, the heuristic component derives a score for each new node as that node is first encountered, and the search of the tree is driven by an agenda of nodes encountered but not yet explored, with that agenda kept sorted by node score. The score for a single node is derived by testing the left-hand sides of all of the h-rules for that node and then executing the right-hand sides of each h-rule whose left-hand side was satisfied. The same unification matcher used in plan tree expansion is also used to execute the h-rule tests, running against a database of node scores and facts about the metaplans instantiated at neighboring nodes. The effect of the right-hand sides of the h-rules is either to set the score for the node or to add to or subtract some increment from it. The rules are designed so that exactly one **set-score** rule applies to each node, and those rules are tested first. The increment rules are then tested, with the resulting score adjusted if necessary to remain within the defined 0..100 range.

Note that while the heuristics and tree building approach have been described in terms of new tree structure being created from the root toward the leaves, Pragma also searches upward over tree structure already instantiated in previous searches. When working with an initial query, where only the top-level plan is known, Pragma does work downward from the root, encountering each node for the first time when it creates it as part of expanding its parent. The heuristic scoring in that case starts at the root, with it receiving the score 100. For subsequent queries, however, the search begins from the context that matched the previous query. With the search information and scores from the previous query cleared away, that single previous context node is marked as encountered and given the score of 100, and the search starts by expanding that node. Expanding in this case means encountering all of a node's unencountered

neighbors, parents as well as children, some of which will in this case already exist, while others may need to be created. Note that the triggering mechanism, in particular, means that a second search across the same tree may need to create different children of some nodes than were created in the first pass, since the determination of which nodes are explored depends in part on the current query. However, the effect of the search process and heuristics are thus made to be equivalent whether the search is across previously built or freshly built tree structure.

8.8 Testing for Links

The final result of the search process in the metaplan tree is testing for links between the partial interpretation of the ill-formed query and the queries predicted from the current context in the tree. The match test for possible links is implemented as a special function invoked when query generating nodes in the metaplan tree are explored that calls the unification prover to search for a match between the set of assertions in the partial interpretation of the query and the tree context, meaning the collected preconditions of the current node and all its ancestors, plus the query operator supplied by the query generating node itself. If a match is found, the newly-identified link node is added to a global list along with its heuristic score, later used in ranking it against other links.

As an additional efficiency move, there is a related but weaker test for the possibility of a match that is applied at each ***build-plan*** node. In this test, the query operator is ignored, since the metaplan branch has not yet modeled any choice of query operator, but the rest of the preconditions in the partial interpretation are matched against those available in the ***build-plan*** context. Using the trigger mechanism, the result of this test is used to control the expansion of query-generating metaplan nodes from this particular ***build-plan*** node, since a failure in this test implies that no query generated from this node could possibly link with the current partial interpretation.

The sorted list of identified links is the result returned from the entire search. If more than one possible previous context is to be taken into account, the lists of links from each search are merged. That sorted list then gives Pragma's heuristic judgment as to the most likely locations in the plan tree to have motivated the agent's new query, and each link node also contains a possible replacement for the wildcard element(s) in the partial interpretation of the query based on its explanation of the agent's likely plan-building intent.

8.9 Integration with the Janus NL System

The final major topic in characterizing the coverage of the implementation is describing the place of the Pragma modules in the larger context of the natural language system for which they were designed. While the theoretical approach for Pragma was envisioned in the context of a complete system including a parser, interpreter, and application system, the focus in the implementation has of course been on the modules directly concerned with modeling plan context and linking from further utterances to the plan model, rather than on the other modules of such a complete system. Nevertheless, maximum possible use in development and testing of the

implementation has been made of the availability of the Janus [57] natural language interface system, so that the Pragma modules were designed to accept a logical representation derivable from the World Model Language (WML) logical form used in Janus and the examples tested in Pragma have been derived from the Janus WML parses for those utterances. While there is an initial version of a translator to convert from the WML into the set of atomic sentences form used for Pragma input, the Janus system does not yet have a functioning scheme for wildcard parsing for ill-formed inputs. Thus the input to Pragma at the moment is taken from WML parses of correct sentences, passed through the translator, with the wildcards then introduced by hand.

Thought was also given to using Pragma's plan model as part of a unified, comprehensive Janus discourse model, and that was born in mind in the design, although that integration work remains to be done. The Janus semantic system does maintain a partial discourse model [5] that represents the set of discourse entities created by each utterance, and that model is used to determine possible referents for anaphoric elements and to track the backward-looking and forward-looking centers. That discourse entity model also stores information about the syntactic structures associated with each entity, for use in calculating co-occurrence restrictions in intrasentential anaphora, but it has no representation of the higher-level pragmatic plan structure of the discourse. An important direction for further work would be to integrate the Janus discourse entity model with Pragma's metaplan model, since centering phenomena like topic shifts that the semantic component currently models should also be represented as context shifts in the metaplan model, so that the two levels of model would reinforce each other. However, Pragma currently functions only as an adjunct to Janus, rather than sharing with it an integrated discourse model.

Although not yet fully integrated, the current implementation of Pragma was demonstrated in March, 1988, as part of a Janus demonstration focusing on the research directions and goals of the project. Working in the SPA domain described in Section 9.2.2, Pragma tracked the plan context as Janus answered a couple of well-formed queries and then made suggested corrections for examples including both alias errors and novel usage examples like

What is the preparedness of Fox?

where the word *preparedness* was intended by the agent as a synonym for *readiness* but was unknown to the system. The well-formed examples used for tracking were translated from the Janus WML form to Pragma's atomic sentence set form by the initial version of the translator mentioned previously, but wildcard parses for the ill-formed cases were worked out by hand in advance. The suggested corrections Pragma identified from the link nodes in the plan tree were presented to the user in a popup menu, and if the user indicated that one of them was correct, the wildcard(s) in the WML were replaced with the appropriate logical forms and the query passed off by Pragma to the remaining stages of Janus processing.

CHAPTER 9

RESULTS

9.1 Introduction

During the development of the Pragma system, a corpus of examples in two different domains has been developed and tested. Because Pragma is not currently connected to the domain model or data base of the actual Janus system, not only the domain plans for a given domain but also the actual instances of entities like ships, their types, and the database facts that describe their current status in the world had to be coded by hand in support of each set of examples. This has limited to some extent the number of examples with which Pragma has been tested and the range of conclusions that can be drawn. Nevertheless, the examples that have been run in the two domains do exercise a broad range of the features of both the metaplan model and heuristic system, leading to useful conclusions about the applicability of these methods for ill-formedness resolution.

This chapter presents and analyzes sets of examples from each of those domains. For each example, there are many variables that affect the number and ranking of the possible links identified for the wildcard elements, including at least the following factors:

- the metaplan tree context established by the previous tracking of well-formed utterances in the problem-solving dialogue,
- the actual wildcard partial interpretation of the ill-formed input, and
- the setting of the cutoff threshold score below which searching stops.

The analysis of the examples looks for patterns within those factors in order to draw conclusions about which factors make the different classes of examples more or less suited to ill-formedness resolution using these methods, about the features of the Pragma system design that contribute to more or less effective handling of the different classes, and about the resulting implications for areas where further work would be fruitful.

9.2 Presentation of Example Results

9.2.1 Damaged Vessel Domain Examples

The first set of examples to be presented are taken from a scenario where the agent is responding to the report of a damaged vessel called the Sterett. For this set of examples, Pragma was run in the mode in which dependent variables in the metaplan tree were instantiated using the actual values from the database. The results for these examples, numbered D1 through D5, are presented following a description of the initial context that was assumed preceding each of the ill-formed examples.

9.2.1.1 The Initial Context

The plan context was established in each case preceding the ill-formed query by having the system track the well-formed input

What class is Sterett?

This query that might indicate that the agent was either thinking of replacing the Sterett with another vessel of its same class or perhaps thinking of supplying spare parts and wondering about the damaged vessel's class as part of predicting what parts would be needed. The input representation for that query is shown in Figure 9.1.

```
(def-partial-parse *class-of-sterett*
  sentence "What class is Sterett?"
  vars (?ship ?class)
  wildcards ()
  types ((?ship aegis-cruiser) (?class vessel-class))
  bindings ((?ship sterett))
  pred-list ((vessel-class ?ship ?class))
  query-op (ask-value ?class
                      (vessel-class ?ship ?class))
  ill-formed nil)
```

Figure 9.1: Input Form for *What class is Sterett?*

This form records for the given input sentence the following:

- a name,
- the actual input string,
- a list of which atoms in the logical forms are variables, that is, symbols whose values are either
  - supplied literally in a bindings clause,
  - fixed by application of predicates, or
  - the object of the query

  (the rest of the atoms in these forms being therefore wildcards, function names, or constants),
- a list of the atoms that are wildcards, that is, those variables whose denotation is unknown not because they are the object of the query but

because they come from portions of the input that were assumed to be garbled,

- a list of pairs giving the types in the type taxonomy for each of the variables found in the formulas,
- a list of bindings for those variables that have particular values assigned to them (often abbreviating for convenience the actual form in the WML which achieves the effect of binding **?ship** to **Sterett** by requiring that **(ship-name ?ship "Sterett")**),
- the actual list of atomic formulas that together place sufficient restrictions on the variables to constrain the query target to be the correct answer,
- a form labeled as the **query-op** encoding the type of query, whether a yes-no question about a formula, a request for the list of members of a set, or a request for the value of a variable, and also which variable or form from which atomic formula was the query target, and
- a final flag telling whether or not the input was ill-formed.

The basic root node for this domain is built from the domain plan header **(have-able-group-for-task ?task ?group ?readiness)**, representing the plan class centered around the goal of having a battle group of ships that is capable of performing a particular task, given that the currently-assigned group is at a particular current level of readiness. This style of beginning with a known class of plans depends on the assumption that the expert in this kind of expert advising dialogue is aware in advance of at least the broad goal that the agent is pursuing. As the system tries to track the plan-building intent of the agent's queries, a tree is built of the possible metaplan moves the agent could use to select and refine a particular plan from within the class of those responsive to the particular overall goal. In this case, the first plan class division separates domain plans that restore the readiness of the current group from those that instead replace that group with another already in a higher readiness state. When the system is presented with the well-formed input just discussed,

What class is Sterett?

the tree of metaplan nodes explored includes the ***build-plan*** nodes shown in Figure 9.2.

```
(have-able-group-for-task
          task-201 kennedy-group C3)                    (1)
  (restore-group-readiness kennedy-group C3)            (2)
    (restore-slot-readiness
            cruiser-slot-1 kennedy-group)               (3)
      (replace-ship sterett cruiser-slot-1)             (4)
      (repair-ship sterett)                             (5)
  (replace-group-for-task task-201 kennedy-group)       (6)
```

Figure 9.2: Partial Tree from Matching *What class is Sterett?*

Because the **replace-ship** plan in line (4) includes among its preconditions the **(vessel-class ?damaged-ship ?damaged-ship-class)** form used in that plan in

combination with other preconditions to require that the ?vessel-class of the vessel to be used as a replacement be the same as the class of the damaged vessel, the *ask-pred-value* metaplan using that precondition can create a child for the **replace-ship** node that will link to the given query. The resulting tree models that the agent is asking the query in order to explore the feasibility of using a plan in the **replace-ship** class to replace the individual vessel in the group whose damaged state has reduced the group's readiness for the task.

Naturally, this is not the only place within the plan tree where the class of the damaged vessel serves as a precondition and thus could motivate a query, but being the closest to the point where the search began (and in the absence of more specific heuristics than tree distance), it receives the highest rank of any of the linking nodes, in this case, 70. Note that the number of possible links in the metaplan tree that will be found for any particular query depends greatly on the value of the low heuristic score cutoff used. When that value is set at 50, the value most frequently used in our tests, the link at **replace-ship** is the only one found. If the cutoff is lowered to 1, its lowest value, 6 other links are identified with scores of 45, 35, 30, 30, 20, and 20. In this case, each of these lower-ranked link nodes is attached to a **sail-direct-conventional** plan, where the class of the vessel is referenced as part of computing whether the distance from the damaged ship's current location to a repair site or port is within the non-refueled cruising range for ships of that (non-nuclear) class. (There are multiple such plan nodes because sailing the damaged ship away is part of various subplan classes for repairing or replacing it, some of which have variables with multiple possible instantiations.) This formulation of domain plans predicts, then, that a query as to the class of the damaged ship is more likely to be related to identifying a suitable replacement ship than to sailing the damaged ship away, although both are reasonable possibilities.

The Pragma system uses the nodes that link to a well-formed query like the above as the new plan context for succeeding utterances. If there are more than one of these linking nodes, a flag controls whether the system uses all of them as initial contexts for the next utterance by running the tree search algorithm once for each possible context, or whether it merely takes the highest-ranked linking node and uses only it as the new starting point. Note that there is a tradeoff here between this choice of how many contexts to carry forward and the choice of the heuristic cutoff that controls the depth of each search within the plan tree, in that carrying forward all the previous linked nodes might also require using a higher cutoff to limit the subsequent searches, while carrying forward only one starting point allows a lower cutoff and deeper search. Most of the testing of Pragma has followed the latter path, using a single node to represent the previous context.

9.2.1.2 Example D1: What is the VOCATION of Fox?

The first ill-formed example, *What is the VOCATION of Fox?*, was run assuming the **replace-ship** context described in the previous paragraphs. Like all of these examples, this is a hand-crafted alias error example, where there is a single word in the input string that is not part of the intended meaning but that is, in itself, a word whose lexical and semantic features are known to the system, so that the system has no *a priori* way of determining which word is in error. Frequently, as pointed out in

Section 8.2, the system finds two or more successful wildcard parses in which different words are assumed to be the erroneous one. Nevertheless, we will assume for these examples a single wildcard parse, indicating by the word set in capital letters which word has been wildcarded.

The representation of the partial interpretation for Example D1 is shown in Figure 9.3.

```
(def-partial-parse *vocation-of-fox*
  sentence "What is the VOCATION of Fox?"
  vars (?ship ?x)
  wildcards (+T+ +P+)
  types ((?ship aegis-cruiser)(?x +T+))
  bindings ((?ship fox))
  pred-list ((domain-pred +P+)
             (+P+ ?ship ?x))
  query-op (ask-value ?x (+P+ ?ship ?x))
  ill-formed t)
```

Figure 9.3: Input for Example D1

In the well-formed query from which this one is derived, the word *location* would give rise in the logical form to some variable for the location, a type for that variable, and a predicate relating that variable to the Fox, as in (**location-of Fox ?location**). Here, where the garbled *vocation* has been treated as a wildcard, we still have a variable, **?x**, but its type is represented by the wildcard **+T+** and the predicate relating it to the ship by the wildcard **+P+**. The predicate list in this example also contains the restriction that the **+P+** wildcard be a **domain-pred**; this helps to prevent spurious matches where the (**+P+ ?ship ?x**) unifies with some inappropriate form like a type predicate. A more extensive type system that ascribed types also to predicates would be another way of enforcing this restriction.

Working outward from the **replace-ship** context established by tracking the previous *What class is Sterett?* query and searching for linking nodes to this partial parse with the heuristic cutoff set at 50, Pragma explores an area of the tree that includes the portion shown in Figure 9.4. The search begins with a heuristic score of 100, shown in the left-hand column, assigned to the ***ask-pred-value*** node on line (2) that linked with the *What is the class of Sterett?* query. The parent ***build-plan*** node (1) is scored 95. Its children include ***ask-pred-value*** nodes for each of the other atomic predicates in that domain plan (3-6) and ***sub-action*** nodes like (18) and (20) for the different steps in plans of the **replace-ship** class, spawning subtrees containing at lower levels ***ask-pred-value*** nodes for their own preconditions.

The **replace-ship** domain plan in node (1) introduces a free variable **?rship** to identify which vessel will be used to replace the damaged one. When first introduced into the plan tree, that free variable is bound to a newly-generated constant like **rship#3**, meaning that the model has not yet tracked any move on the agent's part to consider any particular instantiation for that oper variable. Because that constant will not unify with the **?ship** variable bound to Fox in the partial interpretation, none of the ***ask-pred-value*** nodes involving **?rship** at that level (4-6) can be linked with this input. (Since the **?ship** variable in the plan tree is bound to Sterett, nodes (2) and (3)

```
 95: *build-plan*: replace-ship                          (1)
 100:   *ask-pred-value*:
            (vessel-class ?ship ?ship-class)             (2)
 85:    *ask-pred-value*:
            (location-of ?ship ?ship-loc)                (3)
 85:    *ask-pred-value*:
            (vessel-class ?rship ?ship-class)            (4)
 85:    *ask-pred-value*: (readiness ?rship C1)          (5)
 85:    *ask-pred-value*:
            (location-of ?rship ?cur-rship-loc)          (6)
 85:    *instantiate-var*:  ?rship                       (7)
 100:     *pick-value-suggested*: ?rship fox             (8)
 90:        *build-plan*: replace-ship                   (9)
 80:          *ask-pred-value*:
                 (vessel-class ?ship ?ship-class)        (10)
 80:          *ask-pred-value*:
                 (location-of ?ship ?ship-loc)           (11)
 80:          *ask-pred-value*:
                 (vessel-class ?rship ?ship-class)       (12)
 80:          *ask-pred-value*:
                 (readiness ?rship C1)                   (13)
 80:          *ask-pred-value*:
                 (location-of ?rship ?cur-rship-loc)     (14)
 80:          *sub-action*: replace-ship sail            (15)
 75:            *build-plan*: sail                       (16)
 80:          *sub-action*: replace-ship assign          (17)
 85:    *sub-action*: replace-ship sail                  (18)
 80:      *build-plan*: sail                             (19)
 85:    *sub-action*: replace-ship assign                (20)
```

Figure 9.4: Partial Tree for Example D1

also fail.) The ***instantiate-var*** node at line (7) records that one of the agent's possible plan-building moves when her plan contains such a free variable is to move to instantiate it to a particular value. A subplan form of ***instantiate-var*** called ***pick-value-suggested*** covers those instances where Pragma is able to identify from the partial interpretation a particular nominee for the value to which the agent may be instantiating the variable, and in this case, the occurrence of *Fox* in the query, an instantiation that fits the type constraints and preconditions on the **?rship** free variable, allows the system to use the ***pick-value-suggested*** form of ***instantiate-var*** to model in line (8) the agent's possible consideration of Fox as a filler for the replacement ship slot. (As will be seen later, the system must resort to the other subplan of ***instantiate-var***, ***pick-at-random***, when the partial parse does not suggest a value, as happens when that value is itself the wildcard.) A special heuristic rule applies at line (8), triggered by the fresh instantiation to a value also found in the partial parse, which explains why its ranking jumps back up to 100.

The ***build-plan*** node (9) underneath the ***pick-value-suggested*** is like the one in line (1) except that the **?rship** variable is now bound to the Fox, rather than to the generated constant **rship#3**. That means that at this level, the ***ask-pred-value***

nodes (12-14) for the preconditions involving ?rship can now link to the query operator of the partial interpretation, so that we get three possible answers from those three nodes with the suggested values for the wildcards shown in Figure 9.5.

```
At node (12):
   +T+ = class
   +P+ = vessel-class
At node (13):
   +T+ = readiness-rating
   +P+ = readiness
At node (14):
   +T+ = location
   +P+ = location-of
```

Figure 9.5: Three Links from Example D1

As it happens, there is only one other linking node reached by the search when the cutoff is set at 50, a node within the subtree based on the **sail** action of **replace-ship** at a sub-plan for **sail-nuclear** that includes the precondition **(propulsion-type ?ship nuclear)**. A link is found with score 55 to the ***ask-pred-value*** node for that precondition. The system thus returns those four options, with the first three as equally likely top-level possibilities. It is left to a possible post-process to make use of lexical similarity between the lexeme that was wildcarded and lexicalizations of the different possible logical fillers found, using that as an additional source of heuristic information in deciding between them.

If this example is rerun with the cutoff lowered to 1, many more possible links are identified. (With the lowered cutoff, the context query also finds additional link nodes, but that does not affect the results here, since we are using only the highest-ranked initial context node.) The full list, giving the score, plan, and matching predicate for each, is shown in Figure 9.6.

```
80: replace-ship (vessel-class ?rship ?ship-class)   (1)
80: replace-ship (readiness ?rship C1)               (2)
80: replace-ship
       (location-of ?rship ?cur-rship-loc)           (3)
55: sail-nuclear (propulsion-type ?ship nuclear)     (4)
45: sail-direct-conventional
       (vessel-class ?ship ?class)                   (5)
45: sail-with-refueling
       (location-of ?ship ?ship-loc)                 (6)
45: supply-part-by-ship
       (location-of ?supl-ship ?supl-ship-loc)       (7)
30: sail-direct-conventional
       (vessel-class ?ship ?class)                   (8)
```

Figure 9.6: Full List of Links for Example D1

The three ***ask-pred-value*** nodes from the immediate instantiation within **replace-ship** of **rship** to the Fox still lead the list, of course, but a number of further links (5, 6, 8) are also found to nodes within the **sail** subtree besides the **propulsion-type** query located earlier. The **vessel-class** that is a precondition of

sail-direct-conventional (5, 8) is part of computing whether the location of the damaged ship can be reached by the replacement ship without refueling; the **location-of** the ship is important if refueling *en route* has to be arranged (6).

Line (7) shows a interesting case of a link to an entirely different area of the plan tree. The **supply-part-by-ship** plan is part of **repair-ship**, an alternative to the **replace-ship** node which covers all the other possibilities. The link to this different node proposes that the agent has moved from considering replacing the Sterett to instead considering the option of repairing the damage and then retaining Sterett in its slot in the battle group. One of the actions in the repair plan involves transporting the replacement part to the Sterett, and one subplan for that is to use another ship for the task. The ship to use as the supply ship is, in turn, an open variable in that plan, which the system here supposes the agent may have instantiated to the Fox, and the location of the supply ship is of course a precondition related to its availability for the task. That interpretation of the ill-formed query is in fact a plausible but less likely one in the circumstances, since it involves a much larger planning move away from the previous context. In this case, the lowering of the cutoff brings to light only more distant possibilities, ones that seem remote in comparison to those more highly ranked.

9.2.1.3 Example D2: What is the location of FIX?

The second example is much like the first, except that a different word in the utterance is garbled, the name of the vessel, rather than the name of its attribute, and we are looking at the partial interpretation that has selected that garbled word as the wildcard. The partial interpretation in this case is thus defined as in Figure 9.7.

```
(def-partial-parse *location-of-fix*
  sentence "What is the location of FIX?"
  vars (?w)
  wildcards (+V+)
  types ((?w position)(+V+ vessel))
  bindings ()
  pred-list ((ship-location +V+ ?w))
  query-op (ask-value ?w (ship-location +V+ ?w))
  ill-formed t)
```

Figure 9.7: Input for Example D2

Note that in this case the word *location* has been correctly understood as referring to the predicate **ship-location**, since that is the only semantically relevant sense of the word in this domain, and that means that the type of the wildcard that is its first argument can also be restricted to **vessel**, even though the single-word NP that actually refers to that vessel has been treated as a wildcard.

Figure 9.8 gives an abbreviated segment of the tree that Pragma explores in linking to this ill-formed input. Note that in this case, the ***instantiate-var*** node (4) is not expanded by a ***pick-value-suggested***, since there is no value supplied in the partial interpretation, so that the system has no indication of which ship the agent might be intending to use. The result is that ***pick-at-random*** branches are built (5, 8, 11, 14) for every vessel in the database that fits the preconditions in the **replace-ship** plan

```
95: *build-plan*: replace-ship                          (1)
100:  *ask-pred-value*:
          (vessel-class ?ship ?ship-class)              (2)
85:   *ask-pred-value*:
          (location-of ?ship ?ship-loc)                 (3)
85:   *instantiate-var*: ?rship                         (4)
75:     *pick-at-random*: ?rship biddle                 (5)
70:       *build-plan*: replace-ship                    (6)
60:         *ask-pred-value*:
                (location-of ?ship ?ship-loc)           (7)
75:     *pick-at-random*: ?rship wilson                 (8)
70:       *build-plan*: replace-ship                    (9)
60:         *ask-pred-value*:
                (location-of ?ship ?ship-loc)          (10)
75:     *pick-at-random*: ?rship truxtun                (11)
70:       *build-plan*: replace-ship                    (12)
60:         *ask-pred-value*:
                (location-of ?ship ?ship-loc)          (13)
75:     *pick-at-random*: ?rship fox                    (14)
70:       *build-plan*: replace-ship                    (15)
60:         *ask-pred-value*:
                (location-of ?ship ?ship-loc)          (16)
60:         *sub-action*: replace-ship sail             (17)
85:   *sub-action*: replace-ship sail                   (18)
```

Figure 9.8: Partial Tree for Example D2

as a possible replacement vessel. (Not all are shown here.) A *pick-at-random* branch would also have been built for the Sterett itself, since the **replace-ship** plan does not currently test that the replacement ship not be the same as the damaged one, but its readiness condition causes that branch to fail the precondition tests.

This example thus produces link nodes beneath **replace-ship** to *ask-pred-value* queries of the location of the replacement ship for each possible filler for that role that the agent might be considering, and the number of more distant links to the location queries at **sail-with-refueling** and at **supply-spare-by-ship** is also multiplied. While the links from D1 that referred to other predicates like **readiness** and **propulsion-type** are avoided here because **location** needs to be matched exactly, there are still many more resulting links in this case due to the size of the set of possible fillers. Yet this is similar to the uncertainty of a human expert in similar circumstances, when the only restriction is to the membership of a rather large set. Again, lexical similarity measures and other heuristic information may be able to help restrict the set of likely links.

9.2.1.4 Example D3: Is Fox in the RED?

The partial interpretation for the query *Is Fox in the RED?* is shown in Figure 9.9.

```
(def-partial-parse *is-fox-in-RED*
   sentence "Is Fox in the RED?"
   vars (?ship)
   wildcards (+W+)
   types ((?ship aegis-cruiser)(+W+ position))
   bindings ((?ship fox))
   pred-list ((ship-location ?ship +W+))
   query-op (ask-truth (ship-location ?ship +W+))
   ill-formed t)
```

Figure 9.9: Input for Example D3

When this query is run through Pragma, the identified links for the wildcarded word are to assertions about the location of the Fox, a precondition that we have already seen in a couple places in the plan tree, with the links suggesting that *red* is a garbled version of *Med*. What sets this example apart is that it is phrased as a yes/no question, rather than as a WH query. This means that its query operator is different, **ask-truth** rather than **ask-value**, but because the location of the vessel happens to be the wildcard variable that will therefore match with whatever the actual value in the database is, this case ends up being handled much like example D1, with the exception that the query-generating metaplan used beneath the ***build-plan*** node for **replace-ship** once the ***instantiate-var*** is handled is ***check-pred-value*** rather than ***ask-pred-value***.

Because for these examples the system is running in the mode where dependent variables are instantiated to their database values, the resulting link node, as well as identifying a context in the plan tree that supports one possible form of this ill-formed query, also associates with the wildcard +W+ whatever the actual location of the Fox is recorded to be. Only if the agent was correct in her supposition, naturally, would that value actually match the intention behind the wildcard value. Note that a yes/no query like *Is the FIX in the Med?* where the ship name instead of the function value is the wildcard item would interact differently with this database instantiation mode. In that case, the effect would be that the set of possible fillers would be restricted to those that actually meet the agent's stipulation. If the agent's supposition is correct, the effect will typically be to substantially reduce the number of alternative matches that must be considered. However, if the agent is incorrect, the desired link with the instantiation to the Fox will not appear at all, since the location predicate will fail to match the actual value for the Fox. Indeed, if the agent is unlucky enough to choose an empty location, where neither the Fox nor any other vessel is currently located, the system while operating in this mode will be unable to find any link for that particular query. The SPA domain examples were run in the alternate mode, and this issue will be raised again there, and discussed further in Section 9.5.1.

Another issue to note here is the dependence of the linking algorithm on the exact predicate formulation used. While we are assuming that the parser would treat *in the Med* as a simple **location-of** query, what would happen with a query about being

near the Med, where the actual logical form might be considerably more complex? An important area for further work is extending the model so that the system can recognize the logical connection between a related assertion like *near the Med* and the actual form specified in the plan as the literal precondition.

9.2.1.5 Example D4: Does Thorn have on board a SHARE ER-211 relay?

In this next example, *Does Thorn have on board a SHARE ER-211 relay?*, the logical form is more complex, as shown in Figure 9.10, since the wildcard is an adjective that adds additional restrictions to an already complex NP. That additional meaning is coded by the parser into the single wildcard predicate +P+.

```
(def-partial-parse *share-relay*
  sentence
      "Does Thorn have on board a SHARE ER-211 relay?"
  vars (?ship ?rel ?part-num)
  wildcards (+P+)
  types ((?ship destroyer)
         (?rel relay)
         (?part-num part-number))
  bindings ((?ship thorn)(?part-num ER-211))
  pred-list ((part-number ?rel ?part-num)
             (+P+ ?rel)
             (on-board ?rel ?ship)
             (domain-pred +P+))
  query-op (ask-truth (on-board ?rel ?ship))
  ill-formed t)
```

Figure 9.10: Input for Example D4

This example brings out the importance of the **query-op** field in the partial interpretation representation, since there would otherwise be no way of identifying which of the predicates was the subject of the yes/no query.

A portion of the tree expanded in searching for links to this query is given in Figure 9.11. Note that no links are found for this query underneath the **replace-ship** node (2) that encodes the context from the initial query, since the other predicates about spare parts that are part of this query do not appear as preconditions to any of the subplans or actions in the **replace-ship** branch of the tree. The only link in this case is found by moving up from the initial context to the **restore-slot-readiness** node (1) and then down the **repair-ship** branch (9) through **supply-spare-part** (11) and **supply-part-by-ship** (13) to the ***instantiate-var*** (14) and ***pick-value-suggested*** (15) nodes that model the consideration of the Thorn for the slot of the supply ship. Note that the heuristic score of the ***pick-value-suggested*** (15) node is increased by the special boost given when an open variable slot is instantiated to a value found in the partial interpretation that is not yet included in the context model. Note also that the Thorn could not in any case have been instantiated over on the **replace-ship** side, since, as a frigate, it would not have satisfied the preconditions there about being of the same class as the damaged ship.

```
85: *build-plan*: restore-slot-readiness                (1)
95:   *build-plan*: replace-ship                        (2)
100:    *ask-pred-value*:
            (vessel-class ?ship ?ship-class)            (3)
85:     *instantiate-var*:  ?rship                      (4)
75:       *pick-value-suggested*: ?rship fox            (5)
70:         *build-plan*: replace-ship                  (6)
60:           *sub-action*: replace-ship sail           (7)
85:     *sub-action*: replace-ship sail                 (8)
75:   *build-plan*: repair-ship                         (9)
65:     *build-plan*: wait-for-repair                  (10)
65:     *build-plan*: supply-spare-part                (11)
55:       *build-plan*: supply-part-from-depot         (12)
55:       *build-plan*: supply-part-by-ship            (13)
50:         *instantiate-var*: ?supl-ship              (14)
90:           *pick-value-suggested*: ?supl-ship
                                        thorn           (15)
85:             *build-plan*: supply-part-by-ship      (16)
80:               *check-pred-value*:
                      (on-board ?part ?supl-ship)       (17)
```

Figure 9.11: Partial Tree for Example D4

Once the instantiation at node (15) is made, the assertions in the **supply-part-by-ship** plan match each of the assertions *in the partial interpretation* representation with spare linking to the wildcard +P+, as shown in Figure 9.12.

```
(part-number relay-235 ER-211)
(+P+ relay-235)
(on-board relay-235 thorn)
```

Figure 9.12: Matching Assertions for Example D4

Note that it was because it was running in the mode where dependent variables are instantiated from the database that the system was able, as this tree was being expanded, to fill in some of the actual values used in the match. For example, at the **supply-spare-part** node (11), it was able to retrieve from the database the part number of the damaged part as recorded in the casualty report associated with the damaged ship. If there had been more than one part required, a **supply-spare-parts** plan would have been used, where the filler of that role is a set of parts. Again, at the **supply-part-by-ship** node (13), the system retrieved from the database the actual serial number **relay-235** of the particular instance with that part number that was in stock on board the Thorn. If the Thorn had had more than one part of that type on board, separate ***build-plan*** nodes parallel to this one (16) would have been created for each one. This close tie to the actual database values reduces the size of the tree of possibilities while still making use of the implications for plan context of the actual values used in the query. The disadvantage comes in the heuristic dependence on the agent's knowledge being close to the database picture, and the difficulty of modeling possible contexts that conflict with that. Again, this issue is considered further in Section 9.5.1.

This example also shows the importance of the presence of other assertions to the correct identification of the query context. It is interesting to compare this query with one like *What is the location of Fox?* that could be relevant either as part of a **replace-ship** plan with the Fox as the replacement vessel or as part of **supply-part-by-ship** with the Fox doing the supplying. The ambiguity in that case is due to the lack of reference in the query to any assertions that are specific to only one of the plans, and it is resolved by depending more heavily on the previous context and selecting the nearest link node. In this case, the other assertions in the query provide constraints that allow the system to identify the intended context, even though it is not the closest one. This is an example of the general rule that the more context available in the partially-understood query, the easier it is to identify the desired link for it in the plan context.

9.2.1.6 Example D5: How many BRUISERS are C1?

This last example from the damaged ship domain, *How many BRUISERS are C1?*, includes a sample of a set type variable and requires the use of the ***add-boolean-constraint*** and ***ask-cardinality*** metaplans from the group of metaplans that model plan-building moves that are part of gathering information to support the choice of a particular value in a later ***instantiate-var*** node. The partial interpretation representation for this query is given in Figure 9.13.

```
(def-partial-parse *how-many-bruisers-are-c1*
  sentence "How many BRUISERS are C1?"
  vars (?vset +C+)
  wildcards (+C+)
  types ((?vset (set vessel))(+C+ vessel-class))
  bindings ()
  pred-list ((vessel-class ?vset +C+)
             (ship-readiness ?vset C1))
  query-op (ask-set-cardinality ?vset)
  ill-formed nil)
```

Figure 9.13: Input for Example D5

The *how many* query is rendered using the **ask-set-cardinality** operator applied to the variable **vset**, whose type is sets of vessels. Note that readiness in this domain applies only to vessels, so that even though the word *bruisers* is being treated as a wildcard, the parser is able to deduce the proper type for **vset** from the *C1* restriction applied to it. When predicates like **vessel-class** are applied to a set variable, they are given a distributive reading as applying to each member of the set.

A portion of the plan tree searched in this example is given in Figure 9.14. Once again, the search begins at the previous link node (2) and immediately moves up to **replace-ship**. The successful children of ***instantiate-var*** from the previous examples like ***pick-value-suggested*** are not even explored in this case, since the system can predict from the query operator that this example must involve a ***consider-var-as-set*** metaplan like that in line (4) which models the agent's choice to collect data about the set of possible fillers for the replacement ship slot. The ***consider-var-as-set*** metaplan introduces a generated symbol **rship#3** to represent that set.

```
95: *build-plan*: replace-ship                          (1)
100:  *ask-pred-value*:
          (vessel-class ?ship ?ship-class)              (2)
85:   *instantiate-var*:  rship                         (3)
80:     *consider-var-as-set*: rship#3                  (4)
75:       *add-boolean-constraint*:
                (ship-readiness rship#3 C1)             (5)
65:         *ask-set-data* rship#3                      (6)
60:           *ask-set-cardinality*: rship#3            (7)
70:         *add-boolean-constraint*:
                  (vessel-class rship#3
                                aegis-cruiser)          (8)
60:           *ask-set-data* rship#3                    (9)
55:             *ask-set-cardinality*: rship#3         (10)
```

Figure 9.14: Partial Tree for Example D5

The *add-boolean-constraint* metaplan is used here to model the agent's use in the explicit description of the set appearing in the query of one of the constraints from the plan preconditions. Note that set queries that do not include all of the applicable preconditions can still be appropriate plan-building moves; the agent in this case could have asked about the total number of cruisers, without adding the readiness restriction, or, more far-fetched, asked about the number of C1 vessels of all classes. There are heuristic principles not yet captured in the system that point out that certain orders for constraint application make more sense than others, for example, that constraints that cut down the size of the set most are usually applied first. Thus, the reason that asking for all C1 vessels seems far-fetched is because the weaker **ship-readiness** restriction has been applied first, ignoring the stronger **vessel-class** restriction that would have cut down the set size much further. Nevertheless, it is clear that the system should account separately for whatever collection of restrictions the agent decides to include in a query, so Pragma models each with its own ***add-constraint*** node. However, to prevent multiple linking branches in the plan tree that differ only in the order of application of a set of ***add-constraint*** metaplans, the system does impose an arbitrary (alphabetical) order on the ***add-constraint*** nodes within any branch.

In this example, the **ship-readiness** constraint (5) is the first to be applied. Immediately under that, we see the system's representation (6-7) of the possible query *How many C1 vessels are there?*, but that does not link with the current partial interpretation, which assumes that the wildcard specified some vessel-class restriction on the set of vessels. That additional restriction is modeled by the ***add-boolean-constraint*** in line (8), so that the query in lines (9-10) does provide matches for each of the assertions in the query representation.

Query metaplans like ***ask-set-cardinality*** or ***list-fillers*** model queries that are supplemental to a particular ***instantiate-var*** metaplan, and they are usually followed in the plan-building by moves (whether or not those moves become explicit to the expert in the form of queries) that make use of the data gathered thereby to select a particular instantiation for the open variable. The agent world knowledge heuristics discussed in Section 7.2.2 could be used to help predict the agent's next moves more

precisely based on the previous set data queries and their answers.

9.2.2 SPA Domain Examples

9.2.2.1 Initial Context Description

A second scenario with which the Pragma system was tested, still within the broad domain of naval force deployment, had to do with the prosecution of SPAs, which might be thought of as Sonar Probability Areas, regions of the ocean where a sonar system has detected sounds which could be from an enemy submarine. "Prosecution" of a SPA involves the deployment of various combinations of aircraft, ships, or submarines to listen for and try to locate the possible enemy sub. The typical aircraft assigned to this mission are VP squadrons, while the most useful surface vessels are those that can tow arrays of microphones, one variety of which are called TAGOS vessels. In this scenario, the commander may make manual assignments of assets to SPAs, asking questions about locations or readiness of the units she is considering deploying, but she also has access to a computer system that can estimate the probability of detection for a particular SPA given a certain assignment of assets and suggest initial assignments of available assets to SPAs.

It was within this SPA domain that Pragma was demonstrated as part of BBN's Janus natural language system. Although the wildcard parsing phase needed to be manually simulated for the demonstration, which limited the system to a preselected set of possible queries, Pragma was able to track the user's plan context through a sequence of well-formed queries in this domain and then, presented with ill-formed examples, to locate matching contexts for them in the metaplan tree that in turn suggested corrections for the wildcard elements. Those corrections in turn were substituted into the WML logical forms for the queries which then progressed through the remaining stages of processing, returning the intended system results.

One major difference between this set of examples and those from the damaged ship scenario is in the way the system is handling dependent variables in the plans as the plan tree is expanded. In the damaged ship examples, the system was running in a mode where such variables were bound as encountered to values found in the database. Thus, in that mode, if the **replace-ship** plan refers to the location of the damaged ship that was one of its arguments, the value for that ship's location was bound to the actual location retrieved from the database. As pointed out previously, that approach allows maximum use in plan tracking of the actual constants used by the agent when they are correct. Thus a query like

Which ships are near Gibralter?

can easily and unambiguously be identified as relevant to the **replace-ship** plan if the location of the damaged ship has already been instantiated to Gibralter (or even near Gibralter, if a means of handling deductively-related assertions is used). However, instantiation to the actual database values can prevent a possible match from being found in cases where the agent's beliefs are not correct. If the agent incorrectly believed the damaged ship was near Sicily, the query

Which ships are near Sicily?

might turn up, but Pragma could not match such a query while running in this mode. In fact, this mode of instantiating from the database also prevents the modeling of any yes/no question whose answer is "No", since the logical specifications for an incorrect state of affairs cannot be matched.

To correct this problem, Pragma was run for these examples in a new mode where such variables were bound not to values from the database but to newly generated constants of the same sort used for representing open variables that the agent has not yet instantiated. This effectively hypothesized for each branch of the plan tree that the agent might be considering that the values for the dependent variables were consistent with the plan preconditions, whether or not that was the actual case in the database. For example, if the agent mentioned the cruiser Bainbridge in a query in a **replace-ship** context, the ***instantiate-var*** node that models the instantiation of the **rship** variable to the Bainbridge would supply a generated symbol like **readiness#4** for the value of **ship-readiness** for the Bainbridge. Because that symbol is treated as a variable, it would match in the precondition testing phase with the **C1** readiness value required of the replacement ship, so that the instantiation would be allowed to continue even if the Bainbridge was actually **C4**, on the grounds that the agent might think it was **C1** and thus be pursuing this plan-building path.

Note that in matching with the partial interpretation these assumed variables that have in effect been instantiated from the preconditions can then still be used where appropriate to suggest possible fillers in cases where a restriction value has been wildcarded. For example, if the query were

List the 1 ships near Sterett?

where the lexeme *1* was the wildcard, the **readiness#4** variable would be bound to **C1** from the preconditions, which in turn could be used to suggest a replacement for the wildcard.

The advantage of this approach is that any plan, even one based on incorrect premises, can be modeled in the plan tree. The related disadvantage is that the plan tree can become much larger, particularly when there is an instantiation to all possible values, since a full branch will be built for every instance of the type, even though only a small number of those may actually be suitable candidates when the actual database values of the preconditions are considered. While the Pragma system at the moment can only operate fully in one mode or the other, the goal is a flexible combination of the two approaches, depending on the metaplan context and on the heuristic likelihood of the agent's awareness of the actual status of the given preconditions. The issue is further discussed in Section 9.5.1.

The initial context for the examples in this SPA domain was created by matching to the well-formed query

Which TAGOS are assigned to SPA 1?

a request that assumes that an initial assignment of assets to SPAs has been made, either manually or by the program, and that the agent is querying the current planned assignment of one particular class of asset, the TAGOS towed-array vessels, to SPA 1 in order to acquire data to be used in evaluating or suggesting rearrangements to the

current assignments. The partial interpretation for this context-setting query is shown in Figure 9.15.

```
(def-partial-parse *which-tagos-assigned-to-spa-1*
   sentence "Which TAGOS are assigned to SPA 1?"
   vars (?a-spa ?tset)
   wildcards ()
   types ((?a-spa spa) (?tset (set-of tagos-vessel)))
   bindings ((?a-spa spa-1))
   pred-list ((assigned-to ?tset ?a-spa))
   query-op (ask-value ?tset
                       (assigned-to ?tset ?a-spa))
   ill-formed nil)
```

Figure 9.15: Context Query in SPA Domain

The variable ?a-spa is bound to SPA 1, while tset is an unbound variable that the query seeks to bind to the set of TAGOS vessels that are assigned-to SPA 1.

A portion of the tree explored in linking to that query is shown in Figure 9.16. The top-level goal in this domain is **respond-to-spas** (2), which includes the two subplans **auto-assign-assets** (3) for approaches that use the program to produce the assignments and **manual-assign-assets** (7) for adding or deleting assignments by hand, possibly following a run of the automatic program. Within **manual-assign-assets** comes **manual-assign-to-spa** for approaches that consider a single SPA from the current set at a time, and within that separate subplans for adding assignments of each class of assets (10, 12, 14) and for querying the probability of detection predicted by the model. Each assignment plan includes an action for querying the number of assets of that type currently assigned (11, 13, 15), as well as actions (not shown here) for making the assignment in the model and for actually deploying the asset to the SPA's location.

In this initial context query, a match is found by following the **manual-assign-to-spa** (8) path and then modeling an instantiation (16) of the variable representing the chosen SPA to SPA-1, with that value being suggested by its occurrence in the query (17). Once the instantiation has been made, a link is found to the *ask-pred-value* node (25) underneath **query-tagos-assignments** (24), modeling that the user is asking about the number of TAGOS vessels assigned to SPA 1 as part of a plan to perhaps assign more manually.

With each of these examples in this section, Pragma was run first with the heuristic cutoff at the default of 50, and then again with the cutoff set down to 1, and the results for each case will be given. When the initial context example is run with the cutoff set to 50, as the tree above shows, a single link node is found with a heuristic score of 55, and that remains true when the cutoff is lowered to 1; although a much larger portion of the tree is then explored, that is still the only successful link.

```
100: *root-plan*: respond-to-spas                          (1)
 95:   *build-plan*: respond-to-spas                       (2)
 85:     *build-plan*: auto-assign-assets*                 (3)
 70:       *build-plan*: consider-asset-class              (4)
 70:       *build-plan*: add-restriction                   (5)
 70:       *build-plan*: run-tradeoff-model                (6)
 85:     *build-plan*: manual-assign-assets                (7)
 75:       *build-plan*: manual-assign-to-spa              (8)
 65:         *build-plan*: query-spa-pd                    (9)
 65:         *build-plan*: assign-sub-to-spa              (10)
 50:           *build-plan*: query-sub-assignments        (11)
 65:         *build-plan*: assign-tagos-to-spa            (12)
 50:           *build-plan*:
                   query-tagos-assignments                (13)
 65:         *build-plan*: assign-vp-to-spa               (14)
 50:           *build-plan*: query-vp-assignments         (15)
 65:         *instantiate-var*: ?a-spa                    (16)
100:           *pick-value-suggested*:
                  ?a-spa SPA-1                            (17)
 90:             *build-plan*:
                     manual-assign-to-spa                 (18)
 80:               *build-plan*: query-spa-pd             (19)
 80:               *build-plan*: assign-sub-to-spa        (20)
 65:                 *build-plan*:
                         query-sub-assignments            (21)
 55:                   *ask-pred-value*:
                        (assigned-to ?subs ?a-spa)        (22)
 80:               *build-plan*:
                       assign-tagos-to-spa                (23)
 65:                 *build-plan*:
                        query-tagos-assignments           (24)
 55:                   *ask-pred-value*:
                          (assigned-to ?tagoses
                                       ?a-spa)            (25)
 80:               *build-plan*: assign-vp-to-spa         (26)
 65:                 *build-plan*:
                         query-vp-assignments             (27)
 55:                   *ask-pred-value*:
                        (assigned-to ?vps ?a-spa)         (28)
```

Figure 9.16: Partial Tree for SPA Context Query

9.2.2.2 Example S1: What is the HEADINESS of Assertive?

This example is similar to example D1 in the damaged ship context in having the wildcard affect the name of the attribute of the ship that is the subject of the query. The partial interpretation representation is shown in Figure 9.17.

```
(def-partial-parse *HEADINESS-of-Assertive*
  sentence "What is the HEADINESS of Assertive"
  vars (?a-tagos ?val)
  wildcards (+P+ +T+)
  types ((?a-tagos tagos-vessel)(?val +T+))
  bindings ((?a-tagos assertive))
  pred-list ((+P+ ?a-tagos ?val)
             (domain-pred +P+))
  query-op (ask-value ?val (+P+ ?a-tagos ?val))
  ill-formed t)
```

Figure 9.17: Input for Example S1

In this domain, since the Assertive is a TAGOS vessel, the links that are found are to relevant attributes of TAGOS vessels for plans related to assigning them to SPAs.

An abbreviated portion of the plan tree for this example using the initial cutoff of 50 is shown in Figure 9.18.

```
90: *build-plan*: assign-tagos-to-spa                    (1)
95:     *build-plan*: query-tagos-assignments            (2)
100:      *ask-pred-value*:
                (assigned-to ?tagoses ?a-spa)            (3)
80:     *instantiate-var*: tagos                         (4)
100:      *pick-value-suggested*: assertive              (5)
90:         *build-plan*: assign-tagos-to-spa            (6)
80:           *ask-pred-value*:
                    (ship-readiness ?tagos rdy#4)        (7)
80:           *ask-pred-value*:
                    (location-of ?tagos loc#5)           (8)
80:           *build-plan*: query-tagos-assignments      (9)
80:           *build-plan*: assign-tagos-in-model       (10)
80:           *build-plan*: deploy-tagos-to-spa         (11)
70:             *ask-pred-value*:
                      (location-of ?tagos loc#6)        (12)
70:             *ask-pred-value*:
                      (speed-of ?tagos speed#7)         (13)
70:             *build-plan*: sail                      (14)
65:               *build-plan*: sail-conventional       (15)
65:               *build-plan*: sail-nuclear            (16)
55:                 *ask-pred-value*:
                     (propulsion-type ?ship prop#8)     (17)
```

Figure 9.18: Partial Tree for Example S1 with Cutoff 50

The search process in this case proceeds by moving up to the **assign-tagos-to-spa** parent (1) of the previous context and then down to an instantiation (4) of the open variable in that plan for the TAGOS vessel to be assigned, an instantiation suggested (5) by the occurrence of *Assertive* in the query. Given that instantiation, there are two direct preconditions of **assign-tagos-to-spa** that can link to the ill-formed query, one for **ship-readiness** (7) and one for **location-of** (8). One of the subactions within **assign-tagos-to-spa** is then the **deploy-tagos-to-spa** plan (11) that models planning for actually sailing the TAGOS vessel to the location of the SPA and using it to try to find the submarine. That **deploy** plan includes two further preconditions (12, 13) that link to this query, one of which repeats the earlier **location-of** and the other of which refers to the **speed-of** the TAGOS vessel. (Note that only the preconditions whose form matches that in the partial interpretation and thus that can link to it are being shown.) Finally, there is one further precondition within the deeper plan of **sail-nuclear** that is also reached with this cutoff, a test of the **propulsion-type** (17) of the TAGOS vessel in considering sailing it to the SPA location. Thus, with the cutoff at 50, a total of 5 links are identified, with scores ranging from 80 to 55.

When the cutoff is lowered to 1, many more distant link nodes are identified, bringing the total number to 26, ranked from 80 to 5. A few of the newly identified links come from searching more deeply within the **sail** action of **deploy-tagos-to-spa**, where, as in example D1, **sail-direct-conventional** includes a **vessel-class** precondition and **sail-with-refueling** includes a **location-of** precondition. However, the majority of the newly identified links are found in alternate branches of **manual-assign-to-spa** in which the tree SPA variable is bound to other SPAs than SPA 1. The relevant tree fragment is shown in Figure 9.19.

```
55: *build-plan*: manual-assign-assets                (1)
60:   *build-plan*: manual-assign-to-spa              (2)
65:     *instantiate-var*: ?a-spa                     (3)
70:       *pick-value-suggested*: ?a-spa spa-1        (4)
75:         *build-plan*: manual-assign-to-spa        (5)
55:       *pick-at-random*: ?a-spa spa-2              (6)
45:         *build-plan*: manual-assign-to-spa        (7)
55:       *pick-at-random*: ?a-spa spa-3              (8)
45:         *build-plan*: manual-assign-to-spa        (9)
```

Figure 9.19: Additional Partial Tree for Example S1

Recall that the link node for the initial context query was found underneath the ***pick-value-suggested*** node (4) that bound the SPA variable to SPA 1. All the links for the current query that were found when the cutoff was set to 50 were within that branch, since the alternative instantiations to SPA 2 and SPA 3 (the current SPA database contains 3 SPAs) on lines (6) and (8) involved heuristic scores below the cutoff. However, when the cutoff is set down to 1, the search does explore those alternate branches deeply enough to model instantiation of the TAGOS variable to Assertive in the context of assigning TAGOS vessels to those other SPAs and thus finds successful link nodes there as well. The context mechanism thus behaves in this case as one would expect, with links within the previous context in which SPA 1 is the focus receiving higher scores than the more remote links that assume the agent has meanwhile changed her focus of attention to a different SPA in the domain.

9.2.2.3 Example S2: What is the detection probability for SPA I?

In the query *What is the detection probability for SPA I?*, the question of SPA instantiation is central, since the wildcard item is the *I* that the agent, thinking of the Roman numeral, entered for SPA 1. The partial interpretation representation is given in Figure 9.20.

```
(def-partial-parse *detection-probability-for-SPA-I?*
  sentence
    "What is the detection probability for SPA I?"
  vars (?pd)
  wildcards (+S+ +ID+)
  types ((+S+ spa)
         (+ID+ spa-identifier)
         (?pd prob-of-detection))
  bindings ()
  pred-list ((spa-pd +S+ ?pd)
             (identifier-of +S+ +ID+))
  query-op (ask-value ?pd (spa-pd +S+ ?pd))
  ill-formed t)
```

Figure 9.20: Input for Example S2

Because of the way SPAs are identified in the logical representation, the single wildcard in this query results in two wildcards in the logical form, one for the SPA itself and one for number which is its identifier.

The same combination of locality effects and heuristic cutoff predominates in this case as in the previous query, so that with the cutoff at 50, the only link found is that to the **query-spa-pd** node for the current SPA 1, which was the instantiation inherited from the previous context. When the cutoff is lowered, the system then also identifies link nodes within the other instances of **manual-assign-to-spa** where the SPA is instantiated using ***pick-at-random*** to each of the other SPAs in the database, giving 3 additional link nodes. Because of the dependent variables mode being used for these examples, the system also explored the path where the free variable for the SPA being considered was left set to the generated symbol, and there was also a link in this case to that branch. While such links do not provide any suggested corrective information for the wildcard, they do model the possibility that the agent may by mistake have intended a SPA ID number that does not correspond to any SPA in the actual database, a possibility that the system does need to keep in mind.

One factor limiting the number of link nodes in this particular example was the requirement that the **spa-pd** predicate, which occurs only seldom in the tree, be matched literally. It is an important fact in this effort to make maximal use of partial information that a single element of the partial interpretation sometimes will be rare enough in the tree of possible plan-building moves that it alone is able to limit radically the number of possible links.

9.2.2.4 Example S3: What is the SAW readiness of Assurance?

In addition to the overall readiness rating between C1 and C5 assigned to each vessel, there are a set of mission readiness ratings with three-letter abbreviations that describe its current ability to perform particular missions like mobility (MOB), anti-submarine warfare (ASW), and anti-surface warfare (ASU). In this example, *What is the SAW readiness of Assurance?*, the name of one of these mission readiness areas is the wildcard item, as shown in Figure 9.21.

```
(def-partial-parse *SAW-readiness-of-Assurance?*
  sentence "What is the SAW readiness of Assurance?"
  vars (?assur ?mrdy)
  wildcards (+W+)
  types ((?assur tagos-vessel)
         (?mrdy mission-readiness-rating)
         (+W+ mission-area))
  bindings ((?assur assurance))
  pred-list ((mission-readiness ?assur +W+ ?mrdy))
  query-op (ask-value ?mrdy
                (mission-readiness ?assur +W+ ?mrdy))
  ill-formed t)
```

Figure 9.21: Input for Example S3

The mission-readiness predicate expresses for a given vessel and readiness area the appropriate readiness rating.

The results for this example again point out the importance of context and the usefulness of functions within the partial interpretation that occur only rarely in the plan tree. In this case again a single link node is identified when the cutoff is set at 50, an *ask-pred-value* node within the assign-tagos-to-spa plan for the precondition (mission-readiness ?a-tagos asw-mission-area ?m-rdy), because ASW is the only mission area referred to in the preconditions of SPA-related plans. When the cutoff is lowered to 1, three additional link nodes show up that are due to *pick-at-random* instantiation of the SPA free variable to each of the known SPAs, as well as the one to the branch in which the SPA itself is left uninstantiated.

One interesting feature of this example is that while the differences between the identified links are relevant for specifying the new plan context, they make no difference in terms of suggesting possible corrections for the actual ill-formedness, since the corrected meaning representation for this utterance would be the same in each of the contexts. With a rich context model, it is naturally true that many of the distinctions about which information is being stored end up in particular examples to be irrelevant, but the fact that the same issue was crucial in determining the suggested corrections in the previous example (S2) demonstrates the usefulness of such of model.

9.2.2.5 Example S4: Which TAGS are assigned to SPA 2?

In this final example, *Which TAGS are assigned to SPA 2?*, the query suggests a change in the agent's focus from the previous query's SPA 1 to SPA 2. That change in focus is clear in the query itself, shown in Figure 9.22, since the wildcard affects the class of asset assigned, but not the SPA to which it is assigned, which is specifically bound to SPA 2.

```
(def-partial-parse *TAGS-assigned-to-spa-2*
  sentence "Which TAGS are assigned to SPA 2?"
  vars (asset-set a-spa)
  wildcards (+C+)
  types ((asset-set (set-of asset))
         (a-spa spa)
         (+C+ asset-class))
  bindings ((a-spa spa-2))
  pred-list ((assigned-to asset-set a-spa)
             (class asset-set +C+))
  query-op (ask-value asset-set
                (assigned.to asset-set a-spa))
  ill-formed t)
```

Figure 9.22: Input for Example S4

On the first pass, with the search cutoff set to 50, Pragma is unable to find any link nodes for this query. This is because, as in other examples, that cutoff prevents the search from going outside the previous context branch in which the SPA variable is bound to SPA 1, but unlike the other examples, no link is possible within that previous branch since nothing is found to match the occurrence of SPA 2 in the query.

When the cutoff is lowered on the second pass, the situation is different, as shown in Figure 9.23.

```
70: *pick-value-suggested*: a-spa spa-2                (1)
65:   *build-plan*: manual-assign-to-spa               (2)
60:     *build-plan*: assign-sub-to-spa                (3)
50:       *build-plan*: query-sub-assignments          (4)
40:         *ask-pred-value*:
                (assigned-to ?subs a-spa)              (5)
60:     *build-plan*: assign-tagos-to-spa              (6)
50:       *build-plan*: query-tagos-assignments        (7)
40:         *ask-pred-value*:
                (assigned-to ?tagoses a-spa)           (8)
60:     *build-plan*: assign-vp-to-spa                 (9)
50:       *build-plan*: query-vp-assignments          (10)
40:         *ask-pred-value*:
                (assigned-to ?vps a-spa)               (11)
```

Figure 9.23: Partial Tree for Example S4

The system now does reach the new *pick-value-suggested* node (1) that models the instantiation of the SPA to SPA 2, and within that branch, as expected, 3 link nodes are found, one for each of the three possible classes of assets, TAGOS, VP, and subs. The link nodes (5, 8, 11) come beneath the assign-X-to-spa children of manual-assign-to-spa (2), and all three have a score of 40.

This example raises an important theoretical question concerning the ability of the system to model alternate perspectives or approaches in the same plan domain. Note that while the domain plans used by Pragma here group together all the asset assignments for a particular SPA, an alternative perspective would group together all the assignments of a particular class of assets to all the SPAs, and that second approach would have had an advantage in handling this example. This issue is discussed further in Section 9.4.

Another issue raised by this example is the desirability of incorporating into the heuristic search control process in some way information about how many link nodes have already been identified. Such information could be used in part to avoid cases where the search stops before any link node has been found, as happened in this example when using a cutoff of 50. This is discussed in more detail in Section 9.5.2.

9.3 Factors that Affect Example Difficulty

This and the following sections now proceed to analyze the behavior of Pragma on these sets of examples, drawing conclusions from these results about the kinds of examples that Pragma can handle best and about the features of the system design that were most important or problematic in handling them. The first dimension on which the example results will be analyzed is to identify factors in the examples that make them more or less amenable to this approach to ill-formedness processing. Three classes of such factors can be seen in these examples, first, those that help to characterize the inherent degree of ambiguity resulting from the given wildcard word, second, those that affect the ability of the metaplan context to limit that ambiguity, and third, factors in the pragmatics of the current query that make it easier or harder to bring the information in the context tree to bear on resolving the ambiguity. Together these factors help to describe for given contexts and errors the chance that pragmatic context will be able to uniquely resolve the error.

9.3.1 Inherent Ambiguity of the Wildcard Word

The first class of factors are those that determine what might be called the inherent ambiguity of the wildcard word, that is, the size of the set of possible resolutions for it. That set is dependent both on the syntactic and semantic constraints from the remainder of the sentence formulated in the partial interpretation and on the constraints of the domain, including facts like the number of frigates in existence. The size of that set is important because a successful resolution of the ill-formedness requires that the pragmatic context provide sufficient additional constraint to identify the intended referent from within that set, and the size of the set from which it must choose is thus one measure of the difficulty of the task that is left to pragmatics. Because the system only needs to consult pragmatics when the syntactic and semantic

information from the rest of the sentence leaves it ambiguous, there is a trivial class of examples where syntactic and semantic information themselves uniquely determine the intended referent of the wildcard. This section considers ways of predicting for ambiguous examples the size of the possible resolution set.

The primary identifiable factor that can be used to predict the inherent ambiguity is the part of speech role that the wildcard element plays in the query. (While the wildcarded word itself is ignored, the successful wildcard parse that generated the partial interpretation implies a syntactic role for it.) Figure 9.24 summarizes the data from these examples.

Ex.	Wildcard	Part of Speech	Possible Links	Identified Links
D1	VOCATION	N, functional	8	4
D2	FIX	NPR	20+	20+
D3	RED	NPR	20+	1
D4	SHARE	ADJ	3	1
D5	BRUISERS	N, entity class	20+	1
S1	HEADINESS	N, functional	8	5
S2	I	Integer identifier	3	1
S3	SAW	N, in NN compound	6	1
S4	TAGS	N, entity class	3	0

Figure 9.24: Summary Results with Wildcard Part of Speech

For each example, the figure shows the part of speech of the wildcard, an estimate of the number of different resolutions that could have resulted if the assertion containing the wildcard had been tested for possible links to every assertion found anywhere in the plan tree, and the actual number of resolutions suggested by links identified in a search from the previous context node limited by a heuristic cutoff of 50.

One striking thing initially about this data is the limited number of parts of speech represented. This is primarily a side effect of the single-word error assumption for alias errors and the fact that pragmatics is not consulted regarding errors resolvable on syntactic and semantic grounds. After any single word in a sentence is wildcarded, the parser then searches for interpretations of the sentence applying all available syntactic and semantic constraints. Only those examples parse where there is some filler for the wildcard slot that fits with the rest of the sentence, and pragmatic context is resorted to only when there are multiple fillers that fit. This explains the preference for open class words among the wildcards that require pragmatic disambiguation, since a closed class wildcard is much less likely to produce a partial interpretation with multiple possibilities than an open class one. There are cases of such ambiguity even among closed class words, for example, the different articles like *the*, *these*, or *those* that could fill the place of the wildcard in

*Deploy all of **** ships.*

However, most single closed class words are determined enough by constraints from the rest of the sentence to be able at least to severely restrict their possible fillers from syntactic context if wildcarded, while almost any open class wildcard will allow for many possible instantiations, as in

Deploy all of the ****.

In addition to the preference for open class words, the naval database domain further restricts the classes of alias errors found. While entity-class nouns and functional nouns abound in this domain, adjectives are not common and verbs are quite rare. These two factors together account for the unusual part of speech distribution of the examples as a whole.

As for the pattern of number of possible links by part of speech, although part of speech is only one of many factors affecting the different results, the table does suggest some connection, mainly in terms of the number of possible fillers for wildcards of a particular syntactic class. Thus, the greatest ambiguity was found in examples D2, D3, and D5, with either a proper noun that could match with any cruiser name or ocean region or a common noun naming an entity class from all the possible ship types. That fits with our intuition that proper nouns and class-naming common nouns are intended to pick out single members of large possible sets. The functional nouns from examples D1 and S1 had 8 possible links in the metaplan tree, the number of attributes of the given entity class relevant to related domain plans. We would expect the relevant functional nouns to be restricted to a smaller set than proper nouns or entity class names. Only a few possible links were found in the remaining cases, 3 for the adjective (D4) and 6 for the noun compound (S3) cases. The most difficult errors are clearly those where the wildcarded word leaves a large set of possible fillers, and the part of speech data indicates that that is more likely to be true of proper nouns and class-naming nouns than of functional nouns, adjectives, or nouns in noun compounds.

9.3.2 Degree of Contextual Constraint

While the previous section considered part of speech as an indication of the size of the semantic space opened up by the wildcard word, we consider factors here that help to predict the degree of constraint that the pragmatic context will be able to supply for different classes of wildcard. The main factor here seems to be the distribution in the plan tree of the possible matches for the wildcard itself in terms of whether they are clustered or widely spread as an *a priori* indication of the likelihood of finding multiple ambiguous link nodes. While these examples do not provide the data for any developed classification scheme of referent occurrence patterns in the metaplan tree, they do point out criteria that are useful for judging example difficulty.

Referring again to Figure 9.24, the most noticeable difference of this type can be seen between example D2, where all of the possible links were also identified as links in the pragmatic context, and examples D3 and D5, which had similarly large numbers of possible links in the plan tree as a whole, but where the contextual search in those cases only identified a single link as relevant. The difference there was because D2 occurred in a pragmatic context containing a free variable, so that the wildcard could match to any possible instantiation of it, while the contexts for D3 and D5 had only single, already-instantiated values for entities of the wildcard class. There was thus a significant difference based on the pattern of occurrence within the tree of the different classes of entities. While the instances of cruiser occurred tightly bunched in the variable instantiation context, so that the context was not able to disambiguate

them, the locations and class names tended to be distinguishable by plan context. This difference seems as much a feature of the context tree, based on which items are determined at what point of moving down its branches, as it is of the semantic class of the entities themselves, and thus probably cannot be predicted by class alone.

This difference in resolving power of the pragmatic context is also visible in the other examples, though more weakly. The context was able to reduce the 8 possible links for the functional nouns in examples D1 and S1 to 4 and 5 respectively, the number of ship attributes that were referenced in plans closely related to the previous context. The ADJ and integer identifier in examples D4 and S2 that had 3 possible links were restricted by context to a single one, while example S4, also with 3 possible links, found no link at all in context due to the failure of the rest of the query to link to any node in the area of the previous context. In comparison to the strong contrast between examples D2 and D3, these seem to be cases where the possible fillers are fairly evenly distributed in the tree, giving the plan context a fairly strong resolving power. The greater the typical dispersion of the possible matches, the more likely it will be that the heuristic search will be able to isolate a single most likely link node, while entities that occur in clustered form will probably not be fully disambiguated by the plan tree approach.

9.3.3 Logical Step Size

The third dimension suggested by these examples by which to characterize example difficulty is a pragmatic feature of the new query's relationship to the previous context and refers to what might be called the step size, or logical distance between the context of the previous and current queries. A rough measure of that distance in the plan tree model is given by the number of metaplan nodes that must be traversed to reach from the one context to the other. The intuition here, of course, is that queries that are closely related to their previous contexts stand a much better chance of finding as highest-ranked the intended link nodes in the plan tree, a better chance than do those that represent a substantial logical jump from the previous context. When the system has to traverse many nodes to reach the correct link, it is more likely either to find a nearer false match or to be stalled by the heuristic cutoff.

The *SPA I* wildcard in example S2 is one example of this. If the previous context of this query was already on the plan tree branch where the SPA variable was bound to SPA 1, then the intended link is found easily, but if that variable is unbound or bound to some other SPA in the previous context, then the proximity information in the tree can no longer supply the desired result. This matches expectations from human performance, where a sequence of queries that makes gradual movement through the plan tree eases the hearer's plan modeling task and allows more cooperative responses than one that makes large jumps. It seems that that is why agents who are making a query that involves a large jump from the previous context will often add data to their utterance to inform the expert of the new context, the kinds of statements modeled by the inform class of metaplans described in Section 6.7.

Thus the predicted applicability of metaplan ill-formedness resolution can be based in part on the kind of wildcard involved, the number and distribution in the plan tree of possible matches, and the logical step size from the previous to current queries

giving the distance over which the search must be sustained to locate the desired link node. The results for each factor seem to fit with our intuitive expectations as to which kinds of ill-formedness and context would be harder to resolve.

9.4 A Theoretical Issue: Modeling Alternate Perspectives

Most of the examples with which Pragma has been tested are cases where the organization of the domain plans in the plan tree seems to model successfully the organization of the agent's approach toward the problem. However, example S4,

Which TAGS are assigned to SPA 2?

is an interesting exception where it seems that the agent is exploiting an alternate plan structure or perspective on the problem, and that Pragma's failure to handle that case well stems from that clash of perspectives. This result raises an important question that suggests possible extensions to the metaplan model to take account of alternate organizational patterns in the domain plans for a particular goal.

In that example, the domain plans are based on a SPA-primary model, assuming that the agent will naturally group together all the asset assignment decisions related to a particular SPA. That organization of the problem is certainly a valid approach, since the effectiveness of the various assets depends heavily on what other assets are assigned (TAGOS vessels, for example, having a hard time telling friendly submarines from enemy ones), and the final criterion for success in this problem is a function of the independent probabilities of detection for each individual SPA.

However, there is an alternative organization that is asset-primary, where the agent would group together the making of all decisions concerning assignments of a single asset class, like TAGOS vessels. This organization, while perhaps less compelling than the SPA-primary one, does have the advantage of highlighting the connections between assignment decisions for a given set of assets as that set is partitioned among the SPAs currently requiring attention. In this example, an agent who had decided to allocate the TAGOS vessels as a first step in manually assigning the assets to the SPAs might well follow a context query about the TAGOS assigned to SPA 1 with a follow-on query about those assigned to SPA 2. But because that problem organization is not the one reflected in the domain plans used by Pragma, the system cannot easily discern the connection between those two moves. Pragma, using its SPA-primary plan tree, has to switch between major SPA-labeled branches to find the correct new context, and in doing so it looses its connection with the asset class being queried about before, so that the search in the new, SPA 2 branch gives no preference to TAGOS queries over queries for any other asset class.

This problem is similar to one noted by Carberry [10] when using a plan context model for resolving intersentential ellipsis. In her model, the query

Who is the teacher of CS200?

was attached by her plan tracking component to a plan for attending a particular section of a course, one of the subactions for earning credit in a course. In the plan tree structure, the instantiation of the course involved happened higher in the tree than the elaboration of the action of attending a particular section that established the relevance

of the query about the professor's identity. If the agent followed that query with an elliptical

CS400?

Carberry's system had the problem that the new plan branch built down from the new instantiation of the course slot did not carry over any information about which attribute of the former course had been queried due to considering which particular subactions. That example also seems to be susceptible to this analysis of alternate organizational schemes, since an attribute-primary rather than course-primary domain plan scheme could easily have been built for plans to collect information about the professors teaching all possible courses first, followed by data about which courses had free space, their meeting time, and the like.

One answer to this problem would be to expand the domain plan library to include plans modeling the alternate organization. In our current example, that would mean an alternate set of subplans for assigning assets to SPAs organized by the class of asset. The previous context node in our example would then also find a link in the **arrange-tagos-assignments** branch, and the search for links to the ill-formed query beginning from that context node would rank the exploration of TAGOS assignments to other SPAs much more highly than exploring assignments of other assets to SPA 1. However, the costs of such a multiple perspective approach implemented as parallel plan trees comes not only in substantially increasing the size of the tree to be maintained but also in complicating the representation of a single plan context with substantial additional ambiguity, since at least until a pattern of queries established which organizational approach the agent was adopting, context nodes would have to be maintained in both branches. Tracking certainly could still proceed, as is currently done when there are alternate initial contexts, by searching from each one and selecting the one that finds the highest-scored link, since the agent's queries will fall more closely together in the branch whose domain plan organization fits the approach that the agent is following than in a branch whose organization is at cross purposes to the agent's. Still, the multiplication of alternatives would make the practical development of that sort of approach to multiple perspectives significantly more difficult than the single perspective approach adapted in Pragma.

This concern with multiple perspectives is in fact a point where plan modeling for natural language connects with deep issues in knowledge representation concerning the simultaneous representation of the same domain under multiple organizational principles and the ability to model shifts between those perspectives and the influences of one perspective on the others. The natural but very difficult goal is to find a representational scheme that carries the full richness of the alternate perspectives in a more compact way than multiplying out the possibilities in parallel trees for each alternative. Still, the overall point remains the importance of representing in the context model as much as possible of the structure behind the agent's activities, both the plan-building metaplan structure and the domain plan organization, since it is that structure that is crucial to understanding and predicting the course of the agent's queries.

9.5 System Design Lessons

Some of the other issues raised in the examples concern features in the implementation of the metaplan model and heuristic component, pointing out the strengths and weaknesses of particular elements of the approach.

9.5.1 Dependent Variable Instantiation from the Database

One implementation issue already noted concerns whether dependent variables introduced as the plan tree is elaborated are instantiated immediately to the values they have in the database or if they are left unbound. Since the damaged ship domain examples were run in the first mode, and the SPA domain ones in the second, these results provide a useful comparison between the two approaches.

By "dependent" variables, we mean plan variables not included in the argument list of the plan but functionally dependent on variables that are arguments or on other dependent variables. For example, a **sail** plan might take the ship as an argument and include dependent variables for that ship's location and cruising speed. The plan preconditions including assertions like **(location-of ?ship ?ship-loc)** would then establish the values of those dependent variables, which might then in turn be passed as arguments to subactions. In building the plan tree, such dependent variables can either be treated extensionally and instantiated immediately to their actual database values or they can be left unbound and treated as unknown values.

This question in regard to dependent variables is distinct from the situation with open variables, variables introduced by plans that are not dependent even indirectly on the argument values of the plan but are instead open to the agent's free choice. Because there is no dependence, automatic instantiation to a particular value from the database is not an option for these variables. In addition, part of the advantage of the metaplan model is explicit modeling of the agent's instantiation decisions for open variables, so their instantiation is under direct metaplan control. Thus the modeling of the instantiation of open variables is quite different from the question here about the treatment of dependent variables as the plan tree is being expanded.

If dependent variables are instantiated, then the facts from the database can be used in trimming the space of plans that need to be considered, since instantiation will prevent the expansion of subplans whose preconditions are found to fail under instantiation. For example the **sail** plan class is effectively partitioned into the two plan subclasses **sail-conventional** and **sail-nuclear**. For any particular ship argument to the **sail** plan, only one of those plan subclasses is appropriate, and when dependent variables are instantiated, the effect will be that only one of those subplan branches will actually be built.

This constraining effect of instantiation is particularly powerful where a ***pick-at-random*** metaplan has fired, since the plan preconditions can then restrict the instantiation branches that will actually be built by filtering out from the full set of variables of the same type as the open one those for which the plan preconditions fail. For example, the **sail-with-refueling** plan contains an open variable for the oiler to be

used for the refueling and preconditions ensuring that the oiler has enough fuel loaded and can make a rendez-vous with the ship requiring it. If the agent asks an ill-formed query where the wildcard could be the name of an oiler and where the rest of the query is compatible with the preconditions of the **sail-with-refueling** plan, the system could build separate ***pick-at-random*** branches for each vessel in the oiler class. However, if the system is running in a mode that instantiates dependent variables, only those branches corresponding to oilers that actually have enough fuel and could make the rendez-vous would be built.

While constraining plan tree growth is the primary benefit, another advantage of instantiation is found when a value that links to a dependent variable is lost in the wildcarding. If the dependent variable in the plan tree to which it is linked has been instantiated, then that instantiation can serve to suggest a possible filler for the ill-formedness. The problem is, of course, that only if the agent knew the correct value will the suggested correction correspond to her intention. For example, the wildcard *CI* for *C1* in *Is the Fox CI?* will find a link that suggests the intended meaning only if the Fox is actually C1, while otherwise the instantiated value from the database will be suggested even though it is not what was intended.

However, there is a related problem that is even more serious. The major disadvantage of the extensional instantiation approach is that the plan tree can then only model plans that the system believes would succeed. This contradicts a basic assumption about the expert advising setting, that the agent, while aware of the plans for the domain, is not assumed to be aware of the facts, since determining them is the purpose of the consultation. It would be a serious flaw in the model if it were unable to model consideration by the agent of a plan that is blocked by a domain fact whose value, for example, may have changed only recently. For example, if the agent thinks that Pearl Harbor has the spare part that Sterett needs, she may ask about how long it would take to fly it out from there. But if that part is not actually in stock, a system that instantiated dependent variables could not even model the query. Clearly, the goal must be for the system to be able to model consideration by the agent even of plans that are actually infeasible. Yet a system running without any constraint from instantiation will build very large plan trees. The need is thus for some way to combine the constraints of instantiation with the representational adequacy of noninstantiation.

When this difficulty was first encountered with Pragma, one possible solution that was explored was allowing the system to run in instantiated mode but adding explicit modeling of incorrect assumptions on the agent's part using an ***incorrect-assumption*** metaplan. In the example from the preceding paragraph, the agent's query about how long it would take to fly the part from Pearl Harbor to the Sterett could be matched if an ***incorrect-assumption*** metaplan was used to model the agent's incorrect belief that the part was in stock in Pearl Harbor. This approach has the advantage that all dependent variables in the tree are instantiated, either to their correct database values or to ones justified by explicit ***incorrect-assumption*** nodes. However, if incorrect assumptions are hypothesized for every dependent value, the resulting plan tree will clearly be much larger even than a normal one where dependent variables are left uninstantiated, since there are many possible incorrect assumptions about each dependent value. Thus, this approach is only an advantage if the use of incorrect assumptions can be limited somehow to cases suggested by the particular query. That seemed to require a scheme for identifying the differences between the

beliefs implied by the query and the instantiated context so as to predict that a link would be possible given certain incorrect assumptions, but that turned out to be a difficult problem.

A more hopeful path for combining the constraints of instantiation with representational adequacy seems to lie in expanding the metaplan model to include a model of the agent's world knowledge. With an explicit representation for how much the agent knows, the system would have the tools to predict which plans the agent might believe to be feasible, even though the system knows them to be infeasible, and to separate them from the plans that the agent can be assumed to recognize are infeasible. The agent knowledge model is what then gives the necessary element of control over using instantiation constraints. One direction toward using an agent knowledge model to add some instantiation constraint on plan growth is embodied in the heuristics proposed in Section 7.2.2 which use a model of agent knowledge to influence the heuristic scores given to new nodes in the plan tree. It seems that including this portion of the model in the heuristic component would allow for a nuanced treatment of degrees of agent knowledge, appropriate to the expert advising situation, where there is no opportunity to build up a detailed model of the beliefs of the particular agent. Thus the agent knowledge heuristics represent our current approach to this question of making use of extensional database knowledge to constrain tree growth.

9.5.2 Implications for Heuristic Search Control

Another lesson that suggests possible extensions to the system concerns making use of the partial results of a search for link nodes to help control the progress of the search itself. Currently, the heuristic search control rules depend on tree shape and metaplan context and the contents of the partial interpretation, but they have no way to take account of the progress of the search up to that point. Yet, in example S4, we see that the search was stopped by the fixed heuristic cutoff before discovering any link nodes, while in example D1, the system continued to search for new links to reach the cutoff of 50 even after 3 links ranked 80 had already been found.

The obvious alternative approach would be to implement a best-first search, expecting the highest-ranked link node to be found first and then stopping. However, because of the complexity of the heuristic rules, the search does not proceed monotonically downward in scores. Discovery of an instantiation that matches a constant in the partial interpretation, for example, substantially increases the ranking of a node and thus of its children. Thus one cannot assume that the first link node found would end up being the highest-ranked one if the search were continued. That is why the heuristic system was first designed to run for a fixed score distance, ignoring the partial results of the search. But these examples point out the importance of using the results of the search so far as one relevant factor in controlling the search. The search could then be driven beyond the normal cutoff in cases where no link has yet been found, and perhaps cut off earlier in cases where many candidates already exist and it is therefore unlikely (though not impossible) that higher-ranked ones will be found by searching farther. Some mix of the two strategies for search control seems most appropriate, where information about the results so far is used to moderate the fixed-distance search.

9.6 Summary

This chapter has presented a number of examples of the operation of Pragma's plan tracking and ill-formedness resolution strategies from two different domains, the damaged ship and SPA prosecution domains. The results of these example runs were then analyzed to identify factors on various levels that determine the applicability of this pragmatics-based ill-formedness approach for particular classes of examples. The part of speech of the wildcard was seen to be one indication of the semantic space opened up by the ill-formedness, and factors were also discussed affecting the distribution of possible wildcard fillers in the metaplan tree and thus its ability to provide further constraints. Certain of the examples were seen to raise a theoretical question in the design of the model about capturing alternate perspectives that points toward further work in the simultaneous modeling of multiple perspectives over the same space of domain plans. The results also were analyzed for learnings relating to the design of the metaplan tree and matching code, including the advantages and disadvantages of instantiating dependent variables from the database.

CHAPTER 10

CONCLUSIONS

10.1 Summary

This thesis has presented an approach to making use of pragmatic knowledge for resolving ill-formed input. The class of examples considered was single-word alias errors, selected because they are frequently difficult enough to require pragmatic knowledge to correct or even identify, and yet are easily generated and analyzed.

Much of the research effort went into devising a representation strategy for modeling pragmatic context in the chosen setting of expert advising discourse, where an agent is refining and instantiating a plan to achieve a particular goal. A new plan classification structure was derived that classifies domain plans on the basis of effects, rather than actions, so that plan classes in the resulting hierarchy could stand for the agent's partially-specified plans during the plan-building process. To represent the agent's problem-solving actions, that is, the metalevel actions whose effects are to specify the domain plan, a set of problem-solving metaplans was defined, capturing the agent's possible plan-building and query-generating moves. A ***build-plan*** node in the metaplan tree with a particular domain plan class as its argument, for example, captures the agent's consideration of that class of possible solutions. The surrounding nodes in the metaplan tree characterize the subplan exploration or variable instantiation moves that the agent can use to further refine the plan, and the queries that may be used to gain information relevant to those choices.

That metaplan model of the pragmatic context is applied to an ill-formed query by first using wildcard parsing to derive from the query a set of possible parses encoding the syntactic and semantic constraints from the rest of the sentence for each possible word in the sentence being wildcarded. Each parse in the resulting set then partially specifies a query, but includes logical wildcard elements. The metaplan context tree, whose nodes predict queries related to the previous context, can then be searched for nodes that can be linked to these partial interpretations, thus suggesting corrections for their wildcard elements.

That search must be heuristically controlled with great care, both to limit the area in the context tree that must be searched and to ensure as much as possible that the suggested corrections most closely related to the previous context are identified first. That control is achieved through a substantial collection of heuristic rules that make use of different classes of knowledge to rank the nodes in the metaplan tree, with some based on the shape and metaplan structure of the tree itself and others based on the particular domain plans involved and on a model of the agent's world knowledge, suggesting which plans the agent is more likely to consider and which queries are more likely to be asked. The heuristic rankings derived from the combination of these

different classes of rules are used both to guide the search of the metaplan tree and to rank the possible corrections found, when there are more than one.

The implementation of these techniques in the Pragma system was provided with a library of domain plans in a naval domain and tested with example ill-formed queries based on two different scenarios within that domain. In addition to providing a demonstration of feasibility, the results of the tests also identified factors that affect example difficulty and issues in system design and implementation. Some of the examples from one of the scenarios were included as part of a demonstration of the Janus NL system, loosely integrated with the rest of the Janus system so that the corrected queries could be processed through to answers.

10.2 Implications of this Approach

Although this work was done in the context of a particular class of ill-formedness and a particular discourse setting, there are implications that carry well beyond those limited domains. This kind of model of pragmatic context can have broader usefulness within the expert advising setting, and similar models can be worked out for other discourse settings to generalize the approach still further.

First, then, while the pragmatic model in Pragma is used only to suggest corrections of alias errors, it is a general-purpose model of the task-derived pragmatics for that particular discourse setting, and there are many other uses to which such pragmatic knowledge can be fruitfully put. The handling of many classes of ill-formedness that are not alias errors could also benefit from the use of pragmatic context knowledge, though the style in which to apply that knowledge might vary. Non-alias single-word errors like spelling errors represent a class of ill-formedness that could be treated by the same wildcard parsing approach outlined here. On the other hand, examples of scrambled word order would require alternate methods for bringing the pragmatic knowledge to bear; wildcard parsing in that case would not be an effective way to capture the partial information present in the ill-formed input to prepare for matching against the space of predicted queries from the pragmatic context. Still, even these examples would require the same kind of heuristically-guided prediction of the agent's likely next steps and queries. An effective model of pragmatic context is not only useful in correcting ill-formedness, but would even allow the identification of the ill-formedness in certain examples where nothing in the syntax or semantics betrays the query as ill-formed, based on the absence of any connection between the new query and the previous context.

There are also many possible applications for this kind of pragmatic model in generating cooperative responses. While earlier work has pointed out the usefulness of even a straightforward domain plan model in being able to make cooperative responses to an agent's queries, the additional level of pragmatic structure captured in the metaplans would allow that to be carried much further. For example, the ability to interpret the agent's use of restrictions like *frigates within 500 miles of Sterett* in terms of their role in limiting the size of the set of possible values would allow the system to respond more helpfully when a literal answer would not serve the purpose. Here, if no ships meet the specified restriction, a cooperative response might expand the negative answer with a list of a small number of the frigates closest to Sterett. As another

example, the evaluative comparison context created by tracking a query about the fuel levels for a set of possible ships might suggest that the expert also inform the agent about their fuel consumption rates if those were different enough that the fuel level comparison by itself would be misleading.

As is clear from its use in generating cooperative responses, the fundamental usefulness of a model of pragmatic context comes in understanding the intended meaning of the agent's queries. It is recognizing the role of the query in its context that allows the system to respond cooperatively. In fact, the same sort of context is often necessary for interpreting the meaning of the query itself. For example, queries about *ships within 500 miles* in some contexts might refer to the distance as a ship could sail it, avoiding land masses, while in other contexts it might refer to a helicopter's straight line path. Thus even correct literal responses to well-formed queries require the ability to interpret the query in its pragmatic context, the kind of connection that the metaplan model makes possible.

This pragmatic modeling approach could also be extended to other discourse settings by working out the appropriate metaplans to express the discourse structure of those settings. Some work in this direction has already been done by Litman, as pointed out in Section 6.2.3, who used metaplans to model discourse structures like interruptions and clarification subdialogues. Other settings like explanation and argumentation could also be formalized in this way, so that their structure could be modeled by metaplans. New issues raised in extending metaplan modeling to such domains would include the modeling of changes in world state based on partial execution of discourse metaplans, since those settings, unlike expert advising, are not isolated from such effects. In such settings also the questions surrounding alternative metaplan formulations over the same region of domain plans would come up in a somewhat different way than in the expert advising domain. In the expert advising domain, because the final goal of either the normal ***build-plan*** or the ***evaluate-plan*** trees over a single area of domain plans is the selection of a single domain plan, the alternate organizations can be closely connected so that the agent can move back and forth freely. However, in an argument discourse setting, alternative metaplans like ***point-out-benefits*** and ***attack-position-of-opponent*** may be more tightly segregated.

Not only the basic metaplan modeling approach for pragmatic context, but also the heuristics can be applied in other settings or even to other modeling techniques. The heuristics based on a model of the agent's world knowledge, in particular, provide useful controls on plan tree growth in any situation where domain plans are being used to model an agent's possible actions.

This research has thus taken the basic ideas of modeling pragmatic context using plans and metaplans and applied them in a given discourse setting to resolving a particular class of ill-formedness, thus pushing them in new directions concerning the kinds of models to use and the heuristics by which to control them, and developing them further in ways that are broadly applicable to pragmatic modeling for NL understanding.

10.3 Areas for Further Work

Many areas in which this metaplan model and its heuristics could be fruitfully carried further even within the expert advising and response to ill-formedness settings have already been mentioned in the text. Some of the most significant are discussed here.

First, on the level of the domain plans, it would be useful to relax the assumptions of the correctness and completeness of the agent's knowledge of the plans themselves by following Pollack [41] and representing plans in terms of their constituent beliefs about the possibility of actions and about what their effects will be. That would allow the system to recognize actions that come, for example, from exploring flawed plans. If a student who has just found out that there is room in CIS 659 then asks for a blank permission-of-instructor form, the system should be able to recognize that request (perhaps by analogy with another department that does use such forms) as due to a flawed plan, and explain that that step is not necessary as part of the current plan. This extension to recognize flawed plans would require additional heuristic controls, of course, since it is clearly impossible to explicitly include branches for all possible flawed plans in the metaplan tree. Still, it is a serious drawback that the current model cannot respond to any plan not included in its library especially for a system trying to demonstrate robust understanding.

Another important extension on the domain plan level is extending the model to be able to recognize the deductive consequences of facts rather than just the facts themselves. For example, where the plan speaks of the class of the vessel being a fast frigate, the agent might ask whether its hull number begins with "FF", or if it has SPY-1 radar, perhaps, as an indirect way of determining the class. Like the previous example, such uses of related assertions significantly complicate the search for links to the plan context, since successful matching in these cases requires that the system be able to figure out the intended indirect connection between the data requested and the assertion that occurs in the plan. However, that connection may depend on private knowledge of the agent, or at least may only be deducible from shared knowledge by a complex deduction process. This would seem to require, then, that the expert devote considerable resources, as part of the search for a match, to inferencing on the queried assertions searching for such hidden connections between the actual assertions requested and those that occur in the known plans. The depth of that search, on the other hand, could in turn be limited by the knowledge that the agent is trying not only to communicate her queries, but also wants the expert to be able to maintain an accurate model of her domain planning state, and that secondary goal will usually constrain the agent to supply whatever additional data is required for the expert to be able to make the correct connection and recognize the role of the new query in the agent's plan-building process. Recognizing deductive consequences would also be important in maintaining the agent world knowledge model, since the agent could also be expected to recognize to some limited depth conclusions entailed by newly learned facts.

There are also important directions for further research in regard to the metaplan level of the model. Most important, perhaps, would be to develop a more complete account of the effects of alternate simultaneous metaplan organizations

across the same space of domain plans. In the approach given here, that overlap is handled by using alternate branches in the metaplan tree which duplicate the domain plan structure, but that does not allow for as easy movement back and forth between related metaplans as seems to be found in actual dialogues. An agent who has used ***build-plan*** queries to establish the feasibility of a particular branch will often then ask evaluative queries about some subplan, and it does not make sense to have to build a copy of the branch for that purpose. There should be a way to maintain the separate implications for next moves of the different metaplan classes without duplicating tree structure.

An interesting extension to the metaplans in this setting that should be fairly straightforward would be to add a way to recognize queries where the agent asks not about a domain fact that bears on a plan but about the plan itself or conclusions that depend on the plan. For example, the query

Can the Fox replace the Sterett?

directly asks the expert for a conclusion about the feasibility of a particular subplan based on the expert's knowledge of all the preconditions involved. A similar kind of query in an ***evaluate-plan*** context might ask directly

Would replacing or repairing Sterett be faster?

a query that requires a synthesis of considerable data on the expert's part. These kinds of queries go beyond the explicitly database style of expert advising queries modeled in the current effort toward a more general cooperative problem solving model.

It is also important to connect this style of metaplan context modeling with the plan recognition work that tries to identify the agent's plan from observing her actions. While this current effort proceeded under the simplifying assumptions that the agent's top-level goal was already known to the system, it would clearly be better to combine the two lines of investigation, so that plan recognition techniques could be used to identify the context of an agent's queries within a metaplan model of the possible plans and related queries. Such a facility would not only be important in establishing the initial context in a dialogue where none was given, but also might prove useful as an alternate search strategy in the metaplan tree, where a query that found no links in the immediate neighborhood of the previous context might more easily be matched by applying plan recognition techniques than by expanding the tree further.

An important issue having to do with the model as a whole involves capturing interactions between plans. The approach in this current effort settled for spreading out the planning space as a non-interacting tree of possibilities, but there are actually cases both at the domain and metaplan levels where plans at different points in the tree will interact with each other either constructively or destructively. On the domain level, an agent may realize that the same supply ship being used in one plan to serve as a replacement can also be used on the way to transport a spare part, or that the same oiler cannot deliver its one load of fuel to two different vessels in two different locations. On the metaplan level, a single query metaplan may be able to establish a fact that bears on multiple plans. It seems that these interactions need to be modeled in some explicit form, but it is not clear whether recursive metaplans or some other

scheme would be the best approach.

Concerning the heuristic component of this pragmatic model, one important direction for further work is to expand the limited model of the agent's world knowledge outlined here. This applies particularly in settings where there is extended interaction allowing for the development of a richer model of agent knowledge that can take account not only of factual knowledge but also of knowledge of particular domain plans and of habitual problem-solving patterns. Such patterns could be found both on the domain plan level, where some agents might always try certain plans first, and also on the metaplan level, since individual agents will also have patterns in terms of the ways in which they explore the domain plan space and the sorts of queries they are likely to ask.

Finally, extending the agent's world knowledge model could be made part of a general effort to refine the metaplans so as to model problem-solving behavior with a finer grain of detail. For example, the current metaplans directly model queries beneath any ***build-plan*** node about any of the preconditions of that plan. Control based on the agent's world knowledge over the likelihood of that agent to ask each of those queries is relegated now to the heuristic component. But a more detailed model could explicitly capture that the purpose of the query is to learn the given information in order to help to establish the feasibility of the domain plan, so that the agent's world knowledge would then be factored in as explicit preconditions of the new, more detailed metaplans. Much more could also be captured in such a model about the different kinds of preconditions in the domain plans and which are more likely subjects for queries. At the cost of complicating the model substantially, this expansion would allow more precise prediction based not only on the world knowledge of the agent but also on the nature of the precondition and its role in the domain plan being considered.

The general thrust of all these further efforts would be to expand and enrich the pragmatic context modeling mechanism, since it is the knowledge of context and techniques to apply it that will provide the power to handle many unsolved problems in natural language understanding.

BIBLIOGRAPHY

[1] James F. Allen.
A Plan-Based Approach to Speech Act Recognition.
PhD thesis, University of Toronto, 1979.

[2] James F. Allen and C. Raymond Perrault.
Analyzing Intention in Utterances.
Artificial Intelligence 15:143-178, 1980.

[3] James F. Allen.
Recognizing Intentions from Natural Language Utterances.
In M. Brady and R.C. Berwick (editors), *Computational Models of Discourse*, pages 107-166. Massachusetts Institute of Technology Press, 1983.

[4] Damaris Ayuso and Erhard Hinrichs.
Research and Development in Natural Language Understanding as Part of the Strategic Computing Program, Annual Report December 1985 - December 1986: The Syntax and Semantics of a Meaning Representation Language for JANUS.
Technical Report 6522, Bolt Beranek and Newman, 1987.

[5] Damaris Ayuso.
Discourse Entities in Janus.
In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*. 1989.

[6] Lawrence Birnbaum.
Lexical Ambiguity as a Touchstone for Theories of Language Analysis.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1985*, pages 815-820. Morgan Kaufmann, 1985.

[7] Richard R. Burton.
Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems.
Technical Report 3453, Bolt Beranek and Newman, 1976.

[8] Sandra Carberry.
Tracking User Goals in an Information-Seeking Environment.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 59-63. William Kaufmann, 1983.

[9] Sandra Carberry.
Understanding Pragmatically Ill-Formed Input.
In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 200-206. Association for Computational Linguistics, 1984.

[10] Sandra Carberry.
A Pragmatics-Based Approach to Understanding Intersentential Ellipsis.
In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 188-197. 1985.

[11] M. Sandra Carberry.
Pragmatic Modeling in Information System Interfaces.
PhD thesis, University of Delaware, 1985.

[12] Jaime G. Carbonell.
Towards a Self-Extending Parser.
In *Proceedings of the 17th Annual Meeting of the Association for Computational Linguistics*, pages 3-7. 1979.

[13] Jaime G. Carbonell and Philip J. Hayes.
Recovery Strategies for Parsing Extragrammatical Language.
American Journal of Computational Linguistics 9(3-4):123-146, July-December, 1983.

[14] P.R. Cohen and C.R. Perrault.
Elements of a Plan-Based Theory of Speech Acts.
Cognitive Science 3:177-212, 1979.

[15] Robin Cohen.
Investigation of Processing Strategies for the Structural Analysis of Arguments.
In *Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics*, pages 71-75. 1981.

[16] C.M. Eastman and D.S. McLean.
On the Need for Parsing Ill-formed Input.
American Journal of Computational Linguistics 7(4):257, 1981.

[17] R.E. Fikes and N.J. Nilsson.
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.
Artificial Intelligence 2:189-205, 1971.

[18] Richard H. Granger.
FOUL-UP: A Program that Figures Out Meanings of Words from Context.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1977*. Morgan Kaufmann, 1977.

[19] Richard H. Granger.
The NOMAD System: Expectation-Based Detection and Correction of Errors during Understanding of Syntactically and Semantically Ill-Formed Text.
American Journal of Computational Linguistics 9(3-4):188-196, July-December, 1983.

[20] Richard H. Granger, Chris J. Staros, Gregory B. Tayler, and Rika Yoshii.
Scruffy Text Understanding: Design and Implementation of the NOMAD System.
In *Proceedings of the Conference on Applied Natural Language Processing*, pages 104-106. Association for Computational Linguistics, 1983.

[21] H. P. Grice.
Logic and Conversation.
In P. Cole and J. Morgan (editors), *Syntax and Semantics: Speech Acts*. Academic Press, 1975.

[22] Barbara J. Grosz.
The Representation and Use of Focus in a System for Understanding Dialogues.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1977*, pages 67-76. Morgan Kaufmann, 1977.

[23] Barbara J. Grosz.
Utterance and Objective: Issues in Natural Language Communication.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1979*, pages 1067-1076. Morgan Kaufmann, 1979.

[24] B. J. Grosz and C. L. Sidner.
The Structures of Discourse Structure.
Technical Report 6097, Bolt Beranek and Newman, 1985.

[25] Larry R. Harris.
ROBOT: A High Performance Natural Language Interface for Data Base Query.
Technical Report TR77-1, Dartmouth College, 1977.

[26] Phil Hayes and George Mouradian.
Flexible Parsing.
In *Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics*, pages 97-103. 1980.

[27] Philip J. Hayes and Jaime G. Carbonell.
Multi-Strategy Construction-Specific Parsing for Flexible Data Base Query and Update.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1981*, pages 432-439. Morgan Kaufmann, 1981.

[28] K. Jensen, G.E. Heidorn, L.A. Miller, and Y. Ravin.
Parse Fitting and Prose Fixing.
American Journal of Computational Linguistics 9(3-4):147-160, 1983.

[29] Karen Jensen and George E. Heidorn.
The Fitted Parse: 100% Parsing Capability in a Syntactic Grammar of English.
In *Proceedings of the Conference on Applied Natural Language Processing*, pages 93-98. Association for Computational Linguistics, 1983.

[30] Aravind Joshi, Bonnie Webber, and Ralph M. Weischedel.
Preventing False Inferences.
In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 134-138. Association for Computational Linguistics, 1984.

[31] S. Jerrold Kaplan.
Indirect Responses to Loaded Questions.
In *Proceedings of the Second Conference on Theoretical Issues In Natural Language Processing*, pages 202-209. Univerity of Illinois at Urbana-Champaign, 1978.

[32] Henry A. Kautz.
Toward a Theory of Plan Recognition.
Technical Report TR 162, University of Rochester, July, 1985.

[33] Henry A. Kautz and James F. Allen.
Generalized Plan Recognition.
In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32-37. 1986.

[34] Diane J. Litman and James F. Allen.
A Plan Recognition Model for Clarification Subdialogues.
In *Proceedings of International Conference on Computational Linguistics.* Association for Computational Linguistics, 1984.

[35] Diane J. Litman.
Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues.
PhD thesis, University of Rochester, 1985.

[36] Willian C. Mann, James A. Moore, and James A. Levin.
A Comprehension Model for Human Dialogue.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1977*, pages 77-87. William Kaufmann, 1977.

[37] Eric Mays.
Failures in Natural Language Systems: Applications to Data Base Query Systems.
In *Proceedings of the the National Conference on Artificial Intelligence.* William Kaufmann, 1980.

[38] Steven Minton, Philip J. Hayes, and Jill Fain.
Controlling Search in Flexible Parsing.
In *Proceedings of the International Joint Conference on Artificial Intelligence 1985*, pages 786-787. Morgan Kaufman, 1985.

[39] Richmond Thomason (editor).
Formal Philosophy: Selected Papers of Richard Montague.
Yale University Press, 1974.

[40] C.R. Perrault and J.F. Allen.
A Plan-Based Analysis of Indirect Speech Acts.
American Journal of Computational Linguistics 6(3), July, 1980.

[41] Martha E. Pollack.
Inferring Domain Plans in Question-Answering.
PhD thesis, University of Pennsylvania, 1986.

[42] Lance A. Ramshaw and Ralph M. Weischedel.
Problem Localization Strategies for Pragmatics in Natural Language Front Ends.
In *Proceedings of the International Conference on Computational Linguistics 1984*. Association for Computational Linguistics, 1984.

[43] Lance A Ramshaw.
A Metaplan Model for Problem-Solving Discourse.
In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*. 1989.

[44] Amir M. Razi.
An Empirical Study of Robust Natural Language Processing.
PhD thesis, University of Delaware, 1985.

[45] Ann E. Robinson.
The Interpretation of Verb Phrases in Dialog.
Technical Report 206, SRI International, 1980.

[46] Ann E. Robinson, Douglas E. Appelt, Barbara J. Grosz, Gary G. Hendrix, and Jane J. Robinson.
Interpreting Natural-Language Utterances in Dialogs About Tasks.
Technical Report 210, SRI International, 1980.

[47] Roger C. Shank, Michael Lebowitz, and Lawrence Birnbaum.
An Integrated Understander.
American Journal of Computational Linguistics 6(1):13-30, January, 1980.

[48] Stuart M. Shieber.
An Introduction to Unification-Based Approaches to Grammar.
Center for the Study of Language and Information, 1986.

[49] E.H. Shortliffe.
Computer-Based Medical Consultations: MYCIN.
Elsevier, 1976.

[50] Candace L. Sidner.
What the Speaker Means: The Recognition of Speaker's Plans in Discourse.
International Journal of Computers and Mathematics 9(1):71-82, 1983.

[51] Candace L. Sidner.
Plan Parsing for Intended Response Recognition in Discourse.
Computational Intelligence 1:1-10, 1985.

[52] Norman K. Sondheimer and Ralph M. Weischedel.
A Rule-Based Approach to Ill-Formed Input.
In *Proceedings of the 8th International Conference on Computational Linguistics*. Association for Computational Linguistics, 1980.

[53] David J. Trawick.
Robust Sentence Analysis and Habitability.
PhD thesis, California Institute of Technology, 1983.

[54] David L. Waltz.
An English Language Question Answering System for a Large Relational Database.
Communications of the Association for Computing Machinery 21(7):526-539, 1978.

[55] Ralph M. Weischedel and Norman K. Sondheimer.
A Framework for Processing Ill-Formed Input.
Technical Report, University of Delaware, 1981.

[56] Ralph M. Weischedel and Norman K. Sondheimer.
Meta-Rules as a Basis for Processing Ill-Formed Input.
American Journal of Computational Linguistics 9(3-4):161-177, 1983.

[57] Ralph M. Weischedel, Edward Walker, and Lance Ramshaw.
IRUS/Janus: Natural Language Interface Technology in the Strategic Computing Program.
Signal 40(12):86-90, August, 1986.

[58] Ralph M. Weischedel and Lance A. Ramshaw.
Reflections on the Knowledge Needed to Process Ill-Formed Language.
Machine Translation: Theoretical and Methodological Issues.
Cambridge University Press, 1987, pages 155-167.

[59] Robert Wilensky.
Planning and Understanding.
Addison-Wesley, 1983.

[60] William A. Woods.
The Lunar Sciences Natural Language Information System: Final Report.
Technical Report 2378, Bolt Beranek and Newman, 1972.